

Query Optimization for On-Demand Information Extraction Tasks over Text Databases

by

Mina H. Farid

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

© Mina H. Farid 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Mina H. Farid

Abstract

Many modern applications involve analyzing large amounts of data that comes from unstructured text documents. In its original format, data contains information that, if extracted, can give more insight and help in the decision-making process. The ability to answer structured SQL queries over unstructured data allows for more complex data analysis. Querying unstructured data can be accomplished with the help of information extraction (IE) techniques. The traditional way is by using the Extract-Transform-Load (ETL) approach, which performs all possible extractions over the document corpus and stores the extracted relational results in a data warehouse. Then, the extracted data is queried. The ETL approach produces results that are out of date and causes an explosion in the number of possible relations and attributes to extract. Therefore, new approaches to perform extraction on-the-fly were developed; however, previous efforts relied on specialized extraction operators, or particular IE algorithms, which limited the optimization opportunities of such queries.

In this work, we propose an on-line approach that integrates the engine of the database management system with IE systems using a new type of view called extraction views. Queries on text documents are evaluated using these extraction views, which get populated at query-time with newly extracted data. Our approach enables the optimizer to apply all well-defined optimization techniques. The optimizer selects the best execution plan using a defined cost model that considers a user-defined balance between the cost and quality of extraction, and we explain the trade-off between the two factors. The main contribution is the ability to run on-demand information extraction to consider latest changes in the data, while avoiding unnecessary extraction from irrelevant text documents.

Acknowledgements

For more than two years, I have been working with my supervisor, Dr. Ihab Ilyas. I would like to thank him for his assistance and support in developing this work. He has been a great help, and I learned a lot from him on the academic, professional, and personal levels. I would also like to thank my readers, Dr. Frank Tompa and Dr. Khuzaima Daudjee, for their participation and feedback.

I would like to thank my family for their continuous support during these years. I could not have done it without you.

Dedication

To my father, Hany Fathy Farid, who always believed in me, my mother, Afaf, whose encouragement and advice keep me going, and my sister and her husband, Mariette and Walid, who are always there for me, everywhere.

Table of Contents

List of Tables	ix
List of Figures	x
1 Introduction	1
2 Background and Related Work	7
2.1 Information Extraction Frameworks	7
2.1.1 Apache UIMA Framework	8
2.1.2 General Architecture for Text Engineering	10
2.2 Related Work in Optimizing Information Extraction Tasks	11
2.3 Preliminaries	19
2.3.1 Data Integration	20
2.3.2 View-based Query Processing and View Matching	22
3 Problem Definition and Proposed Framework	25
3.1 Problem Definition	25
3.2 Data Model	26
3.2.1 Attribute Domains and Lineage	26

3.2.2	T-tables as Relational Wrappers	28
3.2.3	Extraction Views and Joiners	29
3.2.4	Document Retrieval	33
3.2.5	System Timeline	34
4	Query Optimization	35
4.1	Optimizing IE tasks	35
4.2	Plan Enumeration and Selection	37
4.3	Join Predicate Push-down	40
4.3.1	Extraction Output Caching	41
4.4	Cost Estimation	41
4.4.1	Quality Estimation and Propagation	43
4.4.2	Statistics Collection	43
5	Implementation and Technical Details	47
5.1	Extending an RDBMS to Support Information Extraction	47
5.1.1	Supporting Information Extraction Systems	47
5.1.2	Implementing the Data Model	48
5.1.3	System Architecture	52
6	Experiments, Results and Evaluation	56
6.1	Experimental Settings	56
6.2	Comparing to Other Approaches	59
6.2.1	Comparing to the ETL Approach	59
6.2.2	Comparing to SQoUT	61

6.3	System Analysis	63
6.3.1	Optimization Benefit	63
6.3.2	Number of Extracted Attributes	65
7	Conclusion and Future Work	66
7.1	Conclusion	66
7.2	Future Work	67
7.2.1	Utilizing Properties of Specific Information Extraction Algorithms	67
7.2.2	Data Cleaning on Extracted Data	68
7.2.3	Probabilistic Information Extraction	68
7.2.4	Multi-Query Optimization: Caching	69
	References	71

List of Tables

1.1	Company Relation Example	2
3.1	Example IE systems	27
4.1	Database statistics	42
6.1	Computing environment for the experiments	57
6.2	IE systems	57
6.3	Extraction views	58
6.4	Joiners	59
6.5	Possible query plans under SQoUT and our system	62

List of Figures

2.1	UIMA Architecture [15]	8
2.2	An Example of An Aggregate Analysis Engine	9
2.3	A Screenshot of UIMA Document Analyzer	11
2.4	Query Tree Representation	23
2.5	View matching and compensation	24
3.1	(a) Original query (b) Equivalent query (c) Execution plan	32
4.1	Execution plan for query in Example 7	39
5.1	System Architecture	52
6.1	ETL vs. JITE	60
6.2	Average goodness of query plans (log scale)	62
6.3	Query execution time in different settings	63
6.4	Referenced columns	64

Chapter 1

Introduction

We live in an era where data is everywhere. Huge amounts of data are being produced and analyzed every day. Data is important mainly because it includes information hidden in it. The perception of data evolved over time. Data used to be stored on magnetic disks in the form of zeros and ones. These data formed text words and numbers that became readable. By applying text analytics algorithms on the data, information can be extracted by identifying entities in the textual data, such as organizations, person names, and locations. Information extraction took data visualization to a higher level by analyzing information to infer knowledge about entities, and figuring out the relations between them.

Information Extraction (IE) systems analyze unstructured text and produce structured relations. The rich and structured access of information and knowledge gives better insight into the original forms of unstructured data. In the presence of large amounts of unstructured data, inferring knowledge buried in this textual data and co-referencing extracted information become together a lengthy operation that needs to be automated for efficient processing and a better analysis. There has been continuous research effort in information extraction over the past few decades.

In the absence of automatic means of extracting such information, one way to complete such tasks is using keyword search to retrieve documents that are relevant to the query at hand, followed by manually identifying the relevant data in the returned documents. This can be a very expensive process which fails to leverage complex extraction techniques to find semantically

Table 1.1: Company Relation Example

Company Name	Employee Name	Job Title
Apple Inc.	Tim Cook	CEO

relevant documents that do not contain the exact text of the keyword query, or when compiling information from multiple documents. Consider a user who is looking for the name of the current CEO of Apple Inc. Using web search engines, the user query is used to find list of relevant web pages, where users can check and find the information they need. Information extraction tries to directly answer the question asked. The IE system would analyze (relevant) web pages and try to come up with a specific answer. For instance, one Washington Post’s article contains the text “Apple chief executive Tim Cook has not made many changes to the company since taking over as its top executive in late August”¹. From this sentence, the required information can be extracted to directly answer the user query, instead of directing the user to the relevant web pages. The statement above can be used to fill in a relation about companies as shown in Table 1.1.

Extracting knowledge and information provides an evidence-based insight into the data. This knowledge can help business management come up with well-informed conclusions and decisions. Knowledge extraction assists the decision-making process by providing required complex analysis together with evidence from the available data. While some of this data is stored in a structured form, unstructured text documents such as Web pages, email messages, news articles and reports contain rich information that can be very useful if extracted on time. The ability to answer structured SQL queries over this kind of unstructured data allows for more complete analysis and better insights into that data. Example applications that benefit from structured queries over unstructured textual data include reputation management systems, which download Web pages to track the “buzz” around companies and products; comparative shopping agents, which locate e-commerce Web sites and index the products offered; and other information extraction applications, which retrieve documents and extract structured relations from the unstructured text. For example, in business intelligence, a company about to release a product *X* may want to know whether similar products are well received by consumers and find out the average price of

¹<http://www.washingtonpost.com/>

such products. Towards this goal, the company extracts information from online shopping Web sites and aggregates the prices and users' ratings of products that are similar to X .

As helpful information extraction is, it faces many technical challenges. The first challenge is the quality of extracted data. The extracted data might be erroneous due to several reasons. First, the information in the text documents might not be accurate or might be out-dated. Extracting such information may misinform the users. However, the correctness of information in the text documents is hard to verify automatically. IE systems sometimes check for redundancy of information to make sure they are correct and produce relations with higher confidence. In general, most IE systems count on the user selecting trusted information sources. The second problem is the accuracy of the extraction process itself. Many algorithms have been developed to analyze the text and learn rules about how to perform the extraction. None of these algorithms is perfect. The extracted data is expected to include false positives and to miss correct information (false negatives). Another challenge is scalability of the extraction algorithms for larger data corpora. Since the processed text is usually semi-structured or unstructured, extracting meaningful information from text requires performing complex text analytic tasks such as language and vocabulary analysis, part-of-speech tagging, named entity recognition, coreference resolution, text segmentation and classification, regular expression matching, and other algorithms [34]. These algorithms process complex operations on data, and they are time consuming. The efficiency of information extraction systems poses a barrier to scalability and processing large amounts of data. Therefore, this process is usually done offline. Information extraction techniques are applied on large amounts of text to extract the data and transform it into a structured format, after which they load it into a data warehouse. Structured queries using SQL or similar languages can then be posed on the data warehouse whenever needed. This approach is usually referred to as Extract-Transform-Load (ETL). However, in the ETL approach, extraction and querying are decoupled in time and space, which results in several limitations.

First, decoupling in time means that the data stored in the warehouse represents a snapshot of the original document collection at the time of extraction, not at the time of query processing. This may be suitable for static document collections, or for applications that do not require recent and up-to-date results. However, for frequently changing data, this snapshot will become stale quickly and will not accurately reflect the original documents. Updating the warehouse in real time is infeasible due to the large amounts of data and the cost of the extraction process [3].

Decoupling in space results from the fact that extraction is often performed with little knowledge (if any) of the schema or attributes needed for future queries, which leads to an explosion in the space of possible relations/attributes to extract. This in turn causes the extraction process to be prohibitively expensive. To avoid such an explosion, assumptions have to be made to limit the space of attributes to extract, which may limit the ability to answer certain future queries. The link to the original data sources is lost, and only lineage information (if any) that is rendered during the ETL operation is stored in the warehouse.

These limitations in many scenarios trigger the need for closer integration between IE and database engines to allow extracting only timely and relevant data within reasonable run-time constraints. We refer to this approach by the term *just-in-time information extraction*, in which extraction is performed as part of query processing rather than as a separate offline process. This ensures that the extracted information is up-to-date and reflects the state of the underlying text document collection at query time. This approach is usually “query-aware”; the query at hand is analyzed and only the relations and attributes needed by that query are extracted, thus avoiding any unnecessary extraction cost. This approach to information extraction is useful in *mashup tools* (e.g. [2, 32]) which join different Web sources to return answers to a user’s query, usually ordered by some user-defined ranking score [33].

Just-in-time extraction can be implemented using specialized extraction operators on text documents to produce structured tuples. However, previous efforts to implement this approach have many limitations regarding query optimization opportunities. The work in [19, 21, 22] uses specialized extraction and join operators to handle text documents and assumes that any two IE systems either return the same set of attributes or have no common attributes at all, which limits the possible combinations of IE systems to use. Furthermore, the join algorithms work only on two document inputs and cannot be extended or chained since the join operators encapsulate the extraction tasks and do not adopt the composability of relational operators. Other approaches either employ query optimizations that are specific to certain IE techniques and not extensible to others [30], or they rely on the user to decide the IE systems to use as well as the order of execution [31]. A more detailed discussion of the related work can be found in Section 2.2.

The new extraction operators have to be fully integrated into the DBMS, which means they have to be added to the relational algebra, and their interaction with the existing relational operators has to be studied (operator precedence, commutativity, etc.). Furthermore, the query

rewrite framework and the query optimizer have to be modified to incorporate these operators into plan enumeration and selection, determining how to enumerate equivalent plans using the new operators. The advantage of such implementation is that it would explore more optimization opportunities and potentially find the most efficient way to answer a query. However, these drastic changes to the DBMS internals can be very hard given the complexity of today's DBMSs.

This work proposes a light-weight implementation of just-in-time extraction that does not require fundamental modifications to the DBMS. The only modifications needed are changes to the view selection algorithms, and modifying the cost model to take into consideration the new structures. These modifications are considered minimal, since most of the used structures already exist in DBMSs, such as views and table-valued functions. Instead of having specialized extraction operators, this approach integrates information extraction with traditional query processing through view matching techniques. Information extraction systems are modeled as *extraction views*; database views whose data is obtained by running corresponding IE systems on specific document collections, rather than by running SQL queries on relational data. Extraction views leverage the current view infrastructure available in most commercial DBMSs. They can be used in queries in the same way as traditional database views. Encapsulating information extraction in views allows queries to reference data from text documents as well as data that is already stored in relational tables, since both types of data appear to the optimizer as relational objects (tables and views). This design allows the DBMS to exploit well-developed query optimization opportunities that are applicable to relational data, including pushing down predicates, and using different access paths, join methods and join orders. These different optimizations, in addition to which extraction views to use for a given query, constitute the plan space for that query. The query optimizer utilizes a cost model to determine the best plan to answer the query. The cost model considers both the efficiency of the IE systems as well as their extraction quality.

In summary, the proposed approach optimizes on-demand information extraction tasks based on the user preference of performance (in terms of running time) or quality of extracted results. The main contributions made are:

- Proposing extraction views as a solution inspired by the data integration paradigm. Extraction views encapsulate the information extraction tasks and use IE systems as black boxes with minimum exploitation of IE metadata.

- Developing a cost model that reflects the cost and output quality of extraction tasks and their effect on query execution and result quality.
- Defining the statistics necessary to estimate execution cost and quality, taking into consideration the information extraction portion, and showing how to obtain such statistics efficiently and effectively from text collections.
- Extending the query optimizer of a relational DBMS to support the new extraction views, giving it the ability to explore the full optimization space for queries involving such views and choose the best plan based on the defined cost model.

The proposed system is evaluated with extensive experiments over a real-world data set and using state-of-the-art IE systems. This work is organized as follows: chapter 2 highlights some of the related work, along with background topics. chapter 3 formally defines the problem and outlines the proposed approach, and chapter 4 presents our optimization techniques and how cost and quality are estimated. The details of the implementation are discussed in chapter 5. chapter 6 presents the experiments conducted and the results obtained, along with performance analysis. The work is concluded in chapter 7, and possible future work opportunities are discussed.

Chapter 2

Background and Related Work

This chapter discusses some of the related work done in the area of information extraction. We start by discussing the various developed IE frameworks in Section 2.1. Then, in Section 2.2, we introduce some of the previous work in optimizing information extraction tasks. The last section, Section 2.3, introduces some key concepts that are related to this work and will be referred to in the following chapters.

2.1 Information Extraction Frameworks

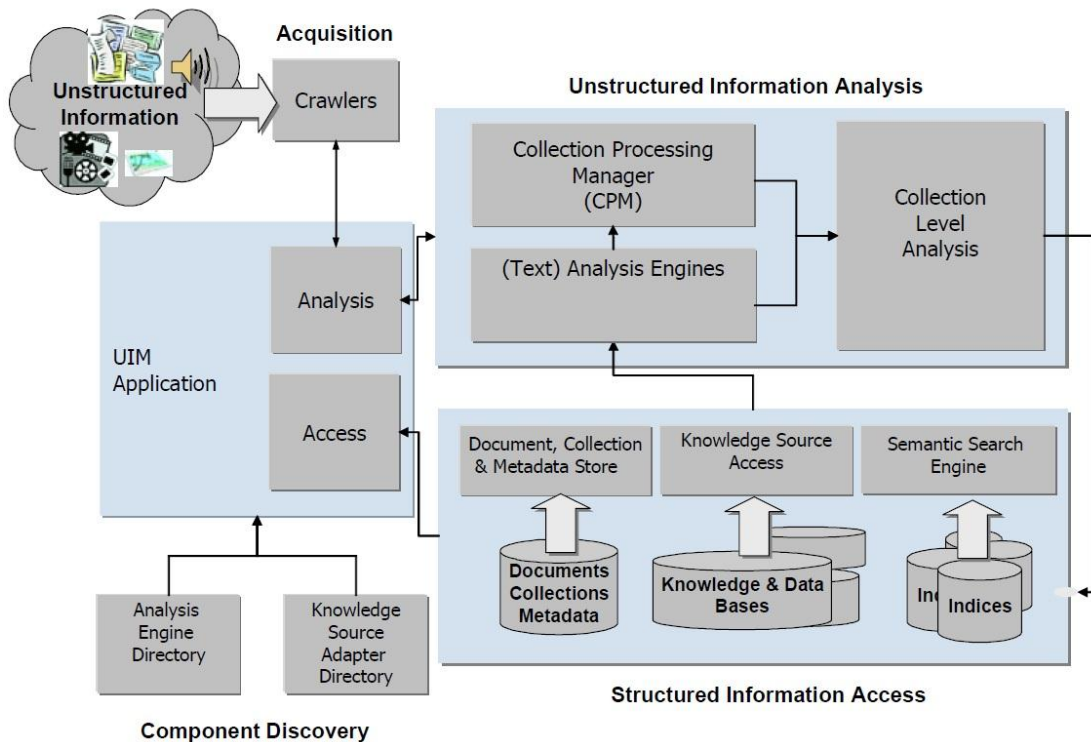
Due to the current explosion in the amount of information being processed and the variety of algorithms developed to perform extraction, some general frameworks have been developed to standardize the process of analyzing unstructured information. These frameworks facilitate the development, debugging, and testing of new information extraction algorithms. A framework acts as a foundation for running information extraction systems. There have been many attempts to develop such frameworks. In this section, two of these frameworks are discussed and briefly explained.

2.1.1 Apache UIMA Framework

Unstructured Information Management Applications (UIMA) is a software framework that supports a development and deployment of multi-modal analytics for processing unstructured information [15]. It is currently an Apache licensed open-source software, and UIMA specifications are standardized by OASIS.

The Apache UIMA framework provides APIs and tools to help the developer build analysis components, such as tokenizers, named-entity recognizers, summarizers, and other components that process unstructured information. It supports both Java and C++ development and allows processing text, audio, and video data. UIMA is capable of wrapping components as network services, and it can scale up to large amounts of data by replicating processing pipelines over a cluster of networked nodes.

Figure 2.1: UIMA Architecture [15]



UIMA Architecture

In many information extraction systems, the complex text analysis process can be decomposed into smaller separate tasks that are performed in sequence. The output of one or more tasks is the input to the next task in the flow. Having this paradigm in mind, UIMA allows developers to divide the data analysis process into smaller, concrete tasks, and every task is modeled separately. The basic modeling unit in UIMA is called an annotator. An annotator contains the analysis logic to perform over data, and it produces a set of annotations, which in turn contains a defined set of features. The input data is analyzed and the annotator marks up the documents according to its logic. For example, an email annotator will produce email annotations over all email addresses found in an input text document. UIMA supports a data structure called the common analysis structure (CAS), which is shared across all elements of the framework. The produced annotations are added to the CAS structure as output annotations. Annotators following in the workflow can utilize previously extracted annotations to perform more complicated tasks. More detailed information about the architecture of UIMA is shown in Figure 2.1

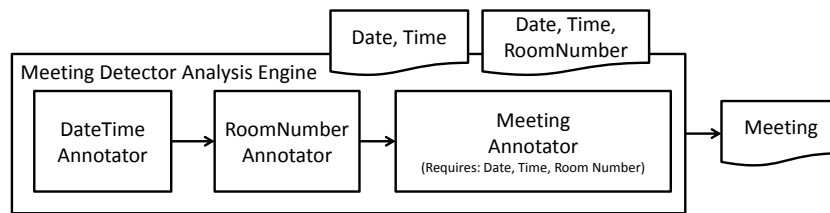


Figure 2.2: An Example of An Aggregate Analysis Engine

Annotators are considered the building blocks of large information extraction systems, which are called analysis engines in UIMA. An analysis engine is responsible for performing complete data analysis tasks. An analysis engine can contain a single annotator, or it can aggregate multiple annotators in a certain sequence. CAS annotations become available for following annotators in the workflow. Figure 2.2 shows an example of an aggregate analysis engine that extracts meetings. It includes three different annotators; DateTime Annotator, which extracts time and date information; RoomNumber annotator, which extracts room numbers according to some regular expression; and Meeting Annotator, which exploits the annotations produced in the two previous analysis engines to infer information about meeting instances.

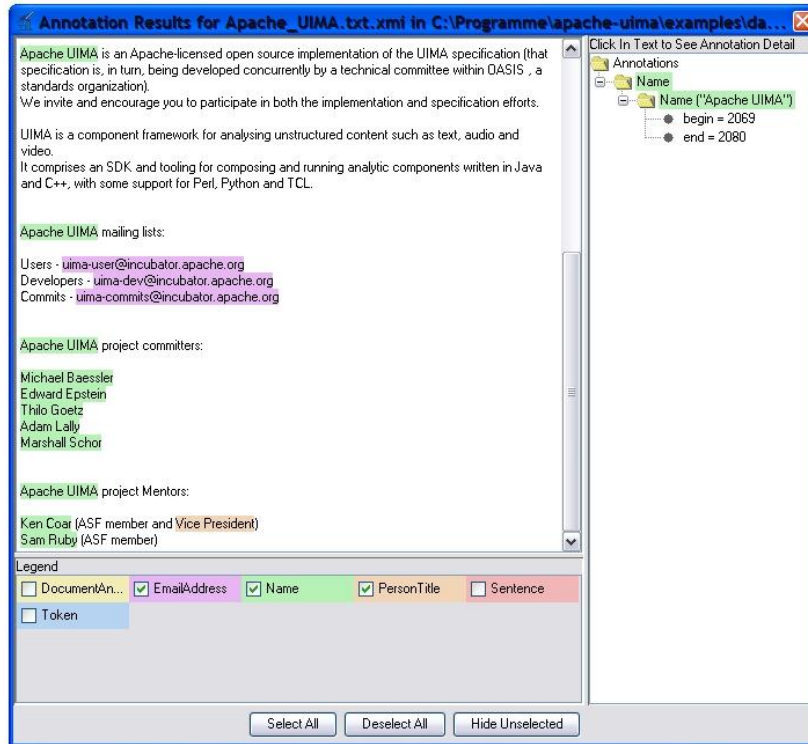
UIMA Tools

UIMA provides a set of useful tools that helps both the developers and users of information extraction systems. The main tool is called UIMA Document Analyzer, which is responsible for testing and running analysis engines on a set of documents. A screenshot of UIMA Document Analyzer is shown in Figure 2.3. It is a GUI that is used to set up the environment required for extraction. Users can select the analysis engine to run, data files to process, and any special encoding. It shows the progress of extraction, and displays the results of extraction. Annotations of processed documents are displayed. The Annotation Viewer is another tool to view stored analysis results. The framework provides a CAS Visual Debugger tool to run UIMA analysis engines and visually navigate through the CAS structure. Another very important tool is the Processing Engine ARchive (PEAR) packager, which is used for distribution and reuse by other applications. The tool help packaging all information, resources, and metadata required for an analysis engine into a PEAR file, and the same tool can be used to deploy a PEAR file on a different machine and install its analysis engine.

2.1.2 General Architecture for Text Engineering

Similar to UIMA, the General Architecture for Text Engineering (GATE) [10] is a framework for developing information extraction systems. The main target of GATE is to separate low-level text processing tasks from language processing algorithms and structures. Also, it provides standard mechanisms to communicate data about language using standards such as Java and XML. The framework is designed to allow reusability of the developed components. Like UIMA, GATE includes some GUI tools to help developing and debugging information extraction modules. A lot of applications have been developed on GATE to do different information extraction tasks such as named entity recognition, coreference resolution, template element construction, template relation construction, scenario template production, etc. GATE is older than 15 years, and has many contributors who work on different projects. GATE also includes many tools to help in the development and deployment of information extraction systems.

Figure 2.3: A Screenshot of UIMA Document Analyzer



2.2 Related Work in Optimizing Information Extraction Tasks

In this section, we will discuss some of the efforts in optimizing information extraction tasks. Researchers have been working on information extraction for many decades. Implementing new extraction algorithms, working on semi-structured, unstructured and web data, developing frameworks for extraction, identifying entities in text documents, extracting relationships between objects, extracting most relevant entities and performing co-reference resolution, extracting probabilistic data with confidence, and other text analytics tasks have been the center of research. We focus on some of the work that is related to our just-in-time extraction (JITE) approach. There have been some attempts to develop generic information extraction systems that allow running structured queries over the extracted data. One of these attempts is System-T that is developed by IBM.

System-T

System-T [24] allows SQL-like queries over text documents. The main focus of System-T is to optimize the performance of rule-based information extraction systems, when running on large volumes of data. The authors present a formal algebraic framework [30, 9] – similar to that used in relational databases – to improve the performance of rule-based extraction tasks. By utilizing IE algebra, System-T performs query optimization and represents extraction tasks by suitable operators to answer SQL-like queries. Data manipulation procedures are also modeled as operators.

System-T includes three main types of operators: relational operators (similar to those of the relational model), span extraction operators (to perform extraction), and span aggregation operators (to aggregate spans). A span is defined as a pair of integers representing the start and end index in a document. System-T handles one document at a time; therefore, a span information is sufficient to identify a word in a certain text document.

Span extraction operators perform rule-based information extraction tasks over text. An operator matches its corresponding rules against the text, and outputs a set of spans that match its input pattern(s). The work in [9] describes an extraction rules language (AQL) and the extraction process properties. There are two span extraction operators; Regular Expression Matcher and Dictionary Matcher. As their names indicate, the regular expression matcher operator compares the document text to some regular expressions, and successful matching spans are added to the output. The dictionary matcher operator can also be thought of like the regular expression matching operators. It takes a list of words (dictionary) and compares the document text to that list of words. The spans of matching phrases are also added to the output.

Span aggregation operators take a set of input spans, and output a set of output spans, after performing some aggregation process on the input set. There are three types of span aggregation operators that correspond to three different span operations; containment consolidation, overlap consolidation, and block operators. A containment consolidation operator rejects spans that are contained within other spans. Overlap consolidation operator merges overlapping spans into one span. The block operator restricts the number of spans within a specified text region. When the block operator finds a violation, it utilizes some rules for selecting which annotations to preserve. The system either keeps the annotation that occurs first, or utilizes a set of tie-breaking rules to

rank the annotation domination.

Since the model of System-T is mainly built upon the traditional relational model, the authors utilize the standard optimization techniques by generating alternative query plans to answer a user query. Moreover, they present three rule-based and text-specific optimization opportunities for the extraction tasks. The first introduced optimization is Shared Dictionary Matching (SDM). SDM is used in Dictionary Matching operators. The operator loads the text token once, loads the dictionaries to match in a hash table, and repeatedly matches the tokens against the hash tables. The main insight is to avoid redundant computation when the same dictionary is used as part of multiple extraction patterns [24]. The second optimization technique is Conditional Evaluation. The idea is to avoid executing a subquery when it is not going to yield any results. For instance, when joining two sub-trees based on a condition, if the first sub-tree does not yield any output, the output of the join will be empty. Therefore, there is no need to execute the extraction of the second sub-tree. The last optimization technique is Restricted Span Extraction. It carefully tries to estimate the extraction span that might yield results, instead of processing unnecessary text. Depending on the semantics of the rules of further attributes, the span of extracted attributes is used, together with the joining conditions, in order to estimate the text span that might yield attributes that are joinable with pre-extracted attributes.

The query optimizer enumerates a space of possible query plans by considering (a) different join orders, (b) the standard transformations like pushing down selections, and (c) additional plans generated by the application of the Conditional Evaluation and Restricted Span Extraction techniques. The optimization in System-T is done on two phases; rule rewriting, and cost-based optimization [24]. In order to produce the same results, rule rewriting is done in a way that does not affect the semantics of the query, but improves its performance by generating more efficient query execution plan. This is done by implementing a more efficient specialized regular expression matching engines. The designed matching engines work for restricted classes of regular expressions. When such a regular expression is to be matched against text, the more efficient engine is used instead of the standard POSIX one. This improves the performance by an order of magnitude [24]. Shared dictionary matching is also considered part of the query rewriting phase. The high selectivity of dictionary look-up tasks is considered a good optimization opportunity that the optimizer utilizes to improve the extraction performance. Cost-based optimization is the core operation of query optimization. The optimizer will consider plans where the con-

ditional evaluation can be utilized to avoid unnecessary extraction. Using these rewritings and optimizations, the optimizer decides on the most efficient query plan based on cost estimates.

System-T is integrated in many systems that IBM provides, such as Lotus Notes (an email client). IBM specialists install appropriate System-T modules according to the business needs. System-T is useful for some applications; however, it suffers from a few limitations. (a) The proposed approach focuses on specific rule-based extraction algorithms. Grammar-based and dictionary look-up might not be practical for several applications that require more complicated text processing tasks. (b) The limited extraction techniques allows extracting only single attributes. In other words, it cannot be used to extract a complete relation. This is due to the fact that the algorithm matches tokens against one rule. This issue is handled by joining individual attributes based on their span properties, constructing an instance or a tuple of a relation, which is not practical if the joining is based on more complex rules or requires further logical analysis. (c) Extensibility is also a problem since these algorithms are integrated inside the operators used for extraction. Moreover, the developer has to write the extraction rules inside the system, disallowing ready made extraction systems to be used [24]. (d) The data model and algebra only represent intra-document operations. The approach is designed to focus on information extraction tasks that process a single document at a time. Therefore, all attributes are expected to exist on the same document, missing the potential to extract attributes from different documents.

SQL Queries over Unstructured Text (SQoUT)

Information extraction systems are used to extract relations from text documents. These structured relations can be queried for complex analysis. Efficiency is a key element in the process of running queries over text databases because extraction is a time-consuming process. Therefore, the query optimizer should choose the most efficient available extraction systems, if possible. Also, it is important to reduce the number of documents that are processed. In general, information extraction does not produce perfect or complete results. It may miss some tuples, or output partially erroneous results. Moreover, ignoring some documents from the extraction process might make the extraction miss some possible results, producing incomplete output. This trade-off between the performance and the quality of the results is discussed in [19, 20, 21, 22, 23]. The authors of [19] proposed a method to consider both information extraction systems quality

of output, and their performance represented in the time taken to perform extraction. This balance is user-specified, i.e., the user chooses which is preferred; quality over performance, or vice versa.

SQL queries over unstructured text (SQoUT) is a system that targets answering SQL queries from unstructured text. The work in [19] splits a table into vertical partitions, such that:

- (a) the key attribute is included in all partitions, and
- (b) no other attributes can exist in more than one partition

Each partition can be extracted by one or more IE systems. The optimizer determines which partitions are relevant to the query, finds the best IE system for each partition, and uses these IE systems to extract the needed partitions, then apply the query on the extracted data. This approach performs better than extracting the whole database, but it has some limitations due to the assumptions made. Since each partition must include the key, an IE system must produce the key plus a number of other attributes. Therefore, it is not possible to extend this system by plugging in and using a general purpose IE system (e.g., an IE system that extracts phone numbers or email addresses). This requires knowledge of the relations to be extracted and their schema when developing the IE systems to use in SQoUT. Also, the join algorithms described in [21] work only on two input partitions, and cannot be extended to more than two, neither can they be chained since they encapsulate the extraction process and do not adopt the composability of relational operators. In terms of determining the best execution plan, the SQoUT project considers data quality as a deciding factor together with execution cost when optimizing the query [21, 20, 23].

When having multiple relations and several databases to extract from, some databases might not be relevant to some relations, and hence, would provide wrong tuples. And since information extraction is a process that takes long time, this time would be wasted when we extract data from irrelevant text corpora. The authors in [23] consider the problem of exploring the potential of a text corpus for extracting data of a certain relation.

The method proposed in [23] determines if a database is relevant to a certain relation. The authors provide a technique that utilizes the tuples extraction confidence score. Since extraction is

not a well-defined process, the extracted data can be false. Some information extraction systems take advantage of various mathematical techniques to estimate the confidence of the extracted data as score. In addition, the authors use the redundancy of extraction to boost the confidence of the extracted tuples. It is likely that a certain fact would be mentioned many times in the same text collection. Extracting data several times gives a confirmation about its validity. Both the extraction confidence score and the redundancy information contribute in calculating the aggregate confidence score of a tuple. This score gives an opportunity to rank the extracted tuples. The confidence of some extracted tuples can give an idea about the relevance of the text corpus to the examined relation.

The process counts on extracting a few good tuples from the text corpus and uses their confidence score to determine the goodness of that corpus for extracting the relation. The simplest way to pick the tuples is by randomly picking a sample of the documents, and extract tuples from this random set of documents. This solution is not practical since any database would probably contain many low-scoring tuples in a random sample of documents. The other approach is to get the top-k ranking tuples. However, this approach is not desirable since in extraction-based scenario, it would require almost complete processing of the database. A related line of work [34] attempts to rank and return the top results based on the estimated data quality.

XLog

XLog [31] takes a different approach – as it uses a Datalog syntax as opposed to SQL – to perform structured querying over unstructured data. It uses embedded extraction predicates to represent information extraction tasks. An extraction predicate can reflect an IE program that may take input and produces tuples in a certain format. It exploits some limited query optimization opportunities, such as pushing down selections, narrowing down text regions from which the extraction happens, and indexing extraction patterns. However, it has a few shortcomings. First, the way the user writes the query determines how it is optimized. The system only uses the extraction predicates entered as part of the query, even if there exist other equivalent predicates that can run faster. Also, in XLog, the predicates often form a dependency graph that determines the order of execution. So, the optimizer may have very little chance of changing join order for example. Pushing down predicates can only be done if the user specifies rules that determine how

certain predicates can be pushed down the execution plan. In summary, a user writing a query has to also tell the system how to optimize that query. It was not clear from the work in [31] how Datalog optimizations can be used with XLog queries. In addition, the system assumes that for a given query, each row of the result must all come from the same document, i.e., no joining of tuples from different documents.

Structured Querying of Web Text

There has been other attempts to run queries over large amounts of textual data. In [7], Cafarella et al. introduce a Datalog-like querying language to run structured queries over documents retrieved from the Web. The authors propose a query system called extraction database (ExDB) that supports SQL-like queries over Web text [7]. The system utilizes a few information extraction systems to extract all possible structured relations from the available text. The IE systems can extract arity-two facts such as `invented(Edison, phonograph)` using TextRunner [4], and hypernymy terms such as `scientist(Einstein)` using KnowItAll [13]. Extracted data is modeled as probabilistic tuples, since extraction systems can be flawed. ExDB runs extractors over the downloaded Web text, and populates the probabilistic data model with all possible information. The data model includes Objects (Einstein, Boston, light-bulb), Predicates (binary tables populated with pairs of objects), and Semantic types which are populated with objects such as `city(Boston)` and `electronics(dvd-player)`. ExDB is designed to support synonymy of objects, predicates, and types. For instance, `invented` and `has-invented` are supposed to be synonyms. Queries are written in a Datalog-like notation. A user can ask a query like `invented(Edison, ?i)` to get a list of all the inventions of Edison. The system keeps query lineage, and represents it by a proof tree for each item in the query's answer. The system also relates to synonym relations that are build using DIRT [27] and includes their results in the query answer.

ExDB provides a good framework for querying web text, but it cannot be used for JITE for the following reasons. First, the extraction is done offline, and all extractors are used to extract all possible relations. This is similar to some extent to the ETL approach, but with a basic difference. In the ETL approach, schemas are defined, and the loading step populates the relations of the schema with the extracted data. In ExDB, however, there is no schema defined.

The system tries to extract all possible relations (uniary and binary relations in this case). In addition, the query language used is still limited, and only SPJU queries are supported. More functionalities need to be implemented.

Extracting Entities from the Web

The work in [34] discusses efficient and effective extraction of concepts from the web. Concepts are entities associated with some attributes. The authors consider both efficiency and quality of extraction. The quality of extraction is affected by the quality of information on the web, and the quality of the information extraction systems used to extract this information. Since extraction is a lengthy process, it is important to avoid extraction from unimportant web documents. The authors in [34] define a robust scoring function for entity attributes extracted from text documents, by casting and addressing this problem as a top-k extraction processing task. The top-k extraction processing algorithm utilizes a scoring function for extracted attribute and returns its top-k values. The scoring function includes the following components: (a) extraction confidence, (b) importance of the webpage (based on its popularity given by a major search engine access log), and (c) the number of sources where extraction originated (how many sources yielded the same tuple, i.e., extraction redundancy). Ranking results is based on the designed scoring function. The extraction confidence is the expected quality of an extracted attribute value given by an IE system. A document importance is entity-specific. A certain document might be important for extracting a certain entity but not important for extracting another entity. This is estimated based on the frequency of accessing a certain document when searching for information related to an entity. This information is retrieved from a search engine log. The authors define an attribute value score as:

$$\text{score}(e, a, D) = \sum \text{importance}(e, d) \times \text{confidence}(e, a, d)$$

for all d in D (all documents in the document collection), where e is the entity to be extracted, a is a value for an e -attribute, and d is the web document.

The algorithm for extracting the top-k values consists of four steps:

- 1) Document Selection: get a batch of unprocessed documents. The paper presents four ways to select which documents to process first, but the main target is to try to get documents that would yield high-ranked extracted attributes.
- 2) Perform extraction using IE system X.
- 3) Calculate top-k; update the rank of the extracted attributes, and re-rank the top-k attributes.
- 4) If top-k are reached, stop. Otherwise, go to step 1 again.

Other Approaches

Other related efforts studied the problem of posing queries over text documents that are already structured, e.g. flat files with comma-separated values, or *(key, value)* pairs. Since the documents are already structured, no information extraction is required (or minimal extraction, consisting mainly of string manipulation). For instance, the work in [8] uses a relational form to express queries on structured text documents, where each document is considered as a tuple containing a set of fields, and all documents in the collection have the same fields. In this work, queries have to specify explicitly which document source it references. In addition, the focus of [8] is on join optimization. The work in [18] also operated on structured files as input. The paper offers a high-level vision of the problem, and only focuses on minimizing file access.

2.3 Preliminaries

Before describing our proposed system architecture to implement JITE, we will briefly outline some of the key concepts that are relevant to this work. Section 2.3.1 discusses the problem of integrating data with different formats from multiple sources into one homogeneous data warehouse. Section 2.3.2 describes how to answer queries given a set of views, and it also discusses materialized views, their effect on improving the query answering performance, and some of the challenging problems that face materialized views.

2.3.1 Data Integration

Real life applications need to process huge amounts of information. Business produces large amounts of data in different formats. Business data includes structured relational data, semi-structured data such as web pages (HTML) or XML documents, or completely unstructured data (emails, reports, reviews, memos, etc.). As mentioned above, analyzing data can help give insight into the data and support the decision making process, by providing conclusions and recommendations. In addition to the different formats of data, not all data resides on the same habitat. Data is usually stored in different sources. To come up with a holistic overview of the data, related data needs to be collected from heterogeneous sources that may have different representations or data formats.

Data integration addresses the problem of blending data from different sources and making it accessible through one homogeneous interface [26]. The main benefits of data integration are:

- (a) It allows a single information access point, which may hide interaction with several data sources.
- (b) Requested information can be evaluated by analyzing data from several sources, giving a more comprehensive answer.

Consolidating data residing in autonomous, heterogeneous data sources provides users with a unified view of the data, known as the *global schema* or *mediated schema* [37]. The global schema provides a reconciled virtual view of the underlying data sources, and users do not know about the schemas or data representation of the local sources [26]. One way to achieve data integration is the ETL approach mentioned in chapter 1, to transform the data from the local schemas at each data source to the global schema and store the transformed data in a data warehouse [17]. Data is extracted from different sources, transformed into a common format, and loaded in a data warehouse that keeps all possible information coming from all of the available data sources. The ETL approach works well when the individual data sources are not frequently updated. Otherwise, the transformed data can get out-of-date quickly, which is not suitable for applications that require fresh or real-time data. For this kind of applications, the global schema is mapped directly to data residing at different local sources, instead of physically storing the

data at a centralized warehouse. The global-to-local mapping provides information about where to get the requested information, making the data and the schema definition loosely coupled.

However, this mapping is not an easy task, since information systems are not by default designed to be integrated. Therefore, to provide a unified data model across all sources, an additional adaptation layer is required on top of these sources [37]. This layer (usually called a “wrapper”) is responsible for accessing data on each source, and providing a common data interface that allows creating views on these sources. Queries posed over the mediated schema are translated accordingly to queries on the local schemas of the original sources. There are two main approaches to implement the mapping: *Local-as-view* and *Global-as-view*.

The Local-as-view (LAV) approach associates each element of the source schemas to a query over the global schema [26]. The definition of each view reflects the information that its local data source can provide. Whenever a new source is added, a view has to be created for that source, and then linked to the global schema [17]. This new source does not alter the definition of the global schema or its relation to the other sources. Answering queries over the global schema becomes a matter of finding the set of views that can provide the required results. The query is reformulated in terms of a set of queries over local sources [26]. Finding which views cover the requested fields, and selecting which views to use is a process that requires complex view matching techniques similar to the ones used with materialized views. Materialized views and view matching will be discussed in Section 2.3.2.

The other way of representing the mapping between the global schema and data sources is the Global-as-view (GAV) approach. GAV defines the global schema as a set of views over the data sources, so each element in the global schema is associated to a query over the local schema [26]. Queries over the global schema would map directly to sources, just like translating a query on a view to a query on base tables. As easy as it is to answer queries, adding a new data source requires that the views of the global schema to be re-written in order to consider the new source.

From the query processing point of view, comparison between the LAV and the GAV favors the GAV approach, since mapping is exact, and queries over the global schema can be directly translated to a set of queries over the local schemas. This translation can be based on simple unfolding strategy. In the LAV, however, it is not immediate to infer how to use the sources to answer queries expressed over global schemas [26]. Sources in LAV contribute partially to

the query answer. On the other hand, modeling the data integration system in the LAV is more flexible than the GAV. In our work, we focus on LAV. We discuss the view matching techniques required to answer queries in LAV data integration systems in the next section.

2.3.2 View-based Query Processing and View Matching

In a system where views are defined, view-based query processing studies computing an answer to a query given a set of views. Views can help improve the query processing performance if there exist materialized views that can provide an answer.

Materialized views are database views whose results are precomputed and physically stored on disk. If a user issues a query over a materialized view, its results can be directly fetched instead of having to be recomputed from the base tables, which helps speed up query execution. Furthermore, if the user poses a query over base tables, with similar predicates to what is used in a materialized view, the optimizer can choose to use the materialized view instead of the base table, thus potentially avoiding to redo complex computations.

Choosing which materialized views to use to answer a query is known as *view matching*. View matching is performed on the internal query representation before plan enumeration. In such models, the query is usually represented as a tree with nodes (often referred to as query blocks) representing base tables, selections, joins, and aggregations, and edges representing data flow. Each block specifies what attributes are produced, and what predicates (if any) are used to filter the outputs. This representation is different from execution plans, since it does not show the join order, join methods, or physical operators to be used.

Example 1. Consider the database tables: Owner (oid, name, country) and Car (cid, make, model, year, owner_id). Also consider the query:

```
SELECT model FROM Owner, Car
WHERE oid = owner_id
AND country = 'Canada'
AND make = 'Toyota'
AND year = 2000
```

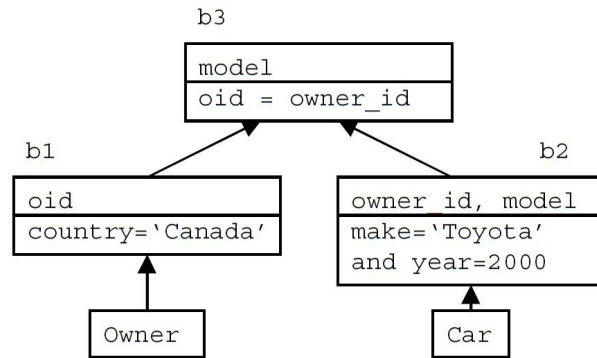



Figure 2.4: Query Tree Representation

Figure 2.4 shows the tree representation of this query. Block b1 represents the selection on the Owner table, b2 represents the selection on the Car table, and b3 represents the join.

Since views are based on queries, they are also represented in the same way. In view matching, the optimizer tries to match a query block with any of the existing views. If a successful match is found, then another query tree is created, using the view instead of the base table(s), and the optimizer is allowed to enumerate plans for both trees, and choose among them based on the estimated cost. Materialized view matching is based on *coverage* or *subsumption*; a materialized view is considered an exact match to a certain query block if the view produces the same tuples and attributes that are produced by that block [36]. In case of non-exact matches, where the view produces more tuples and/or attributes than the query block, a compensation block is added to output only the desired tuples and attributes. This compensation is usually in the form of applying additional predicates, projections, and/or aggregate functions to get only the desired results.

Example 2. Given the tables and query in Example 1, assume that there exists a materialized view v defined as:

$v = \text{SELECT owner_id, make, model FROM Car WHERE year}=2000.$

Figure 2.5(a) depicts the tree representation of v . This view can be used in place of the car table for the purpose of the given query, since it covers the tuples and attributes needed by the query. However, a compensation predicate (block $b2'$ in Figure 2.5(b)) needs to be added.

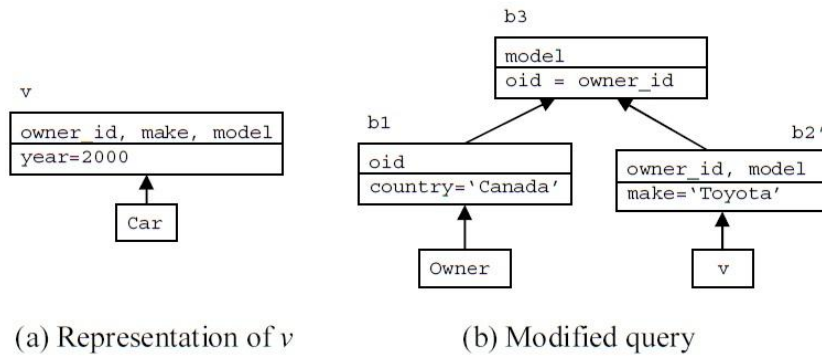


Figure 2.5: View matching and compensation

Adding the compensation operations can potentially be more expensive than using the base table, which is why the optimizer may opt to use the base table even though a materialized view is available. The work in [36] describes the cases and conditions required for materialized view matching.

Chapter 3

Problem Definition and Proposed Framework

We start this chapter by formally defining the approach of just-in-time information extraction (JITE), then we introduce the framework proposed to implement this approach. We describe the data model and its different components, and how they are manipulated to reflect the different operations of JITE.

3.1 Problem Definition

JITE is an approach to extract only query-relevant information online under some performance and quality constraints. We try to avoid assumptions about IE systems and the relations to extract. Since not all documents are relevant to extraction, we want to avoid processing documents that are not likely to yield query-relevant results. The main insight is to be able to perform online extraction to reflect the latest state of data source (data changes), and avoid over-extraction for the relations (and attributes) that are not requested by the query. We explain the data model that we developed to implement JITE. In the following chapters, we will describe how the system utilizes different components of the data model to run queries over unstructured text.

3.2 Data Model

As explained in chapter 1, in the ETL approach, execution and querying are decoupled in time and space, which leads to extracting all possible attributes and relations in advance by invoking all of the available IE systems. In addition to the explosion in the number of attributes and relations to extract, the data extracted soon becomes outdated by the time it is queried. To address these problems, we propose a framework in which information extraction is query-driven. Our proposed framework supports SQL queries on both relational data and data extracted from text documents. Data is extracted from text documents during query processing, based on a schema that is defined on demand.

3.2.1 Attribute Domains and Lineage

Our framework assumes the existence of a set of IE systems \mathcal{E} . The IE systems in \mathcal{E} can be general purpose systems such as UIMA [15] annotators or GATE [10] applications. Each IE system in \mathcal{E} extracts data and returns it as objects with one or more attributes. All objects returned by a given IE system E have the same set of attributes, $attr(E)$. Hence, the output of each IE system can be viewed as a set of tuples sharing the same attributes. It is possible to have multiple IE systems in \mathcal{E} that have overlapping attributes.

Definition 1. *Given the set \mathcal{E} of all available IE systems, the domain universe \mathcal{D} is the set of all possible domains extracted by all IE systems in \mathcal{E} , i.e. for $\mathcal{E} = \{E_1, \dots, E_n\}$, $\mathcal{D} = attr(E_1) \cup \dots \cup attr(E_n)$*

Example 3. *Table 3.1 shows a group of 9 available IE systems, and the domains that can be extracted by each. Based on these IE systems, $\mathcal{D} = \{\text{company, city, name, date, position, color, time, address, country, email}\}$*

The set \mathcal{D} represents the space of extractable information that is accessible to queries. Different applications can use different subsets of \mathcal{D} . The domain names in \mathcal{D} are *unique*, i.e. there can be no two domains with the same name that extract different types of data. For example, all the *company* domains in Table 3.1 refer to the same type of data, and hence *company* exists only

Table 3.1: Example IE systems

IE system	Domain(s)
E_1	company
E_2	company, city
E_3	name
E_4	date
E_5	name, company, position
E_6	color
E_7	time
E_8	address, city, country
E_9	email

once in \mathcal{D} . However, the same domain can be extracted by multiple IE systems. The domain universe \mathcal{D} can be expanded by adding new IE systems that return different domains. These domains are independent of any particular queries or applications.

If an IE system E returns the domains d_1, d_2, \dots, d_m , each tuple returned by E has the following:

- The values v_1, v_2, \dots, v_m , where v_i belongs to the domain d_i ; $1 \leq i \leq m$, and
- The *lineage* of each v_i . The lineage of an extracted attribute value is additional metadata that indicates where that value was extracted from.

The simplest and most common form of lineage information can be $\langle docURL, span \rangle$, where *docURL* is the URL of the document from which a particular attribute value is extracted, and *span* is the position of that value inside the document, usually defined as $(begin, end)$. We assume that all IE systems expose this lineage information in addition to the actual extracted data. In our work, we found that many real-life IE systems do in fact return this information as a minimum [10, 15, 30]. This lineage information is often useful in linking data produced by different IE systems.

3.2.2 T-tables as Relational Wrappers

Our framework supports answering queries that reference both relational data and extracted data. The relational data is represented as traditional relational tables. A relational table consists of (a) an *intension*, which is the table definition (the set of attributes in the table, and the type of each attribute), and (b) an *extension*, the actual data in that table, stored as tuples. To represent the extracted data, we define a special type of table, called *T-tables*.

Definition 2. A T-table consists of only an *intension*, i.e. the table signature describing the attributes in that table and their domains. A T-table has no *extension*, since its data is to be extracted from text documents. Note that a T-table is not a view definition on other base tables.

T-tables provide a relational wrapper to the extracted data in order to be able to pose queries on that data. From the query point of view, a T-table is just like a traditional relational table. The difference is the source of the data in each. In traditional tables, the data has to be loaded before any queries can be posed. This data is stored in a structured form (tuples), and read from disk during query execution. T-tables only have to be defined prior to issuing any queries, but no data is loaded into them. During query execution, data is directly extracted from text documents.

Each T-table can only include data that is available for extraction, i.e. only include attributes that refer to the domains in \mathcal{D} . The T-table definition is in the form $T(a_1 : d_1, \dots, a_n : d_n)$ where each a_i is an attribute name, and d_i is the domain to which a_i belongs.

Example 4. Consider the domain universe from Example 3, and suppose we need to pose queries related to companies and employees. For this particular application, we can define the following T-tables:

- Comp (cname: company, addr: address, cty: city, cntry: country)
- Emp (ename: name, birthdate: date, job: position, ecomp: company, hiredate: date)

Both the *cname* and *ecomp* attributes belong to the *company* domain in \mathcal{D} . Similarly, the *birthdate* and *hiredate* attributes belong to the *date* domain. Note that the T-table *Emp* cannot include a *salary* attribute for example, since we have no *money* domain that can be extracted.

As mentioned in chapter 1, our approach to the information extraction problem is inspired by data integration techniques, specifically the LAV approach. In this sense, our T-tables correspond to the “global schema” or “mediated schema” tables (cf. Section 2.3.1). Defining the T-tables allows the user to pose queries such as:

```
SELECT job, min(hiredate)
FROM Comp, Emp
WHERE cname = ecomp
AND cntry = 'Italy'
GROUP BY job
```

3.2.3 Extraction Views and Joiners

Even though the definition of the T-table specifies the attribute domains, this definition does not specify where the data is obtained from (which document collection) or how to obtain it (which IE systems to use). This is accomplished using *extraction views* and *joiners*.

Definition 3. An extraction view v is defined as $v(T, A, C, E)$, where:

- T is a T-table on which v is defined
- A is the set of attributes in T that v covers
- C is the document collection that is mapped to v
- E is the IE system that will be used to extract the tuples of v from the documents in C

Extraction views are a way of packaging IE systems into logical entities. The extraction view definition tells the system where and how to obtain the data. It establishes the mapping between the T-table attributes, the document collection, and the IE system. Different extraction views can cover different groups of attributes (possibly overlapping) in the same T-table, each using a different IE system and/or a different document collection. An extraction view $v(T, A, C, E)$ is valid only if the following conditions are met:

- All the attributes in A must belong to the T-table T .
- The domains of the attributes in A (as specified in the definition of T) must be among those produced by the IE system E .

In addition to the above conditions, the IE system E must be applicable to the document collection C . This is always true for general purpose IE systems that do not make any assumptions about the document format. However, some extractors may be able to operate only on a specific document format (e.g., binary files, semi-structured and XML documents, or even video files). The user is responsible for assigning the extractors to their appropriate document collections.

Extraction views are similar to traditional database views in that they do not contain materialized data, but have a way of obtaining this data at run time. In traditional database views, the view definition is itself a query. At runtime when a particular view needs to be evaluated, the system executes that view's query and produces the returned data. In extraction views, when a view needs to be evaluated, its corresponding IE system is invoked on the specified document collection, and the extracted information is returned in relational format. Using views as a way to package IE systems allows for arbitrarily complex queries that reference extracted data and relational data at the same time. It also exploits all the database query optimization techniques that are already supported by DBMSs, such as pushing down predicates, and using different join methods and join orders.

Example 5. *Suppose that we have two document collections: C_1 contains company information, and C_2 contains employee information. Given the T-tables defined in Example 4, and given the IE systems in Table 3.1, we can define the following extraction views (among others):*

- $v_1(Comp, \{cname\}, C_1, E_1)$
- $v_2(Emp, \{ecompany\}, C_2, E_1)$
- $v_3(Emp, \{ename\}, C_2, E_3)$
- $v_4(Emp, \{birthdate\}, C_2, E_4)$
- $v_5(Emp\{ename, ecompany, job\}, C_2, E_5)$

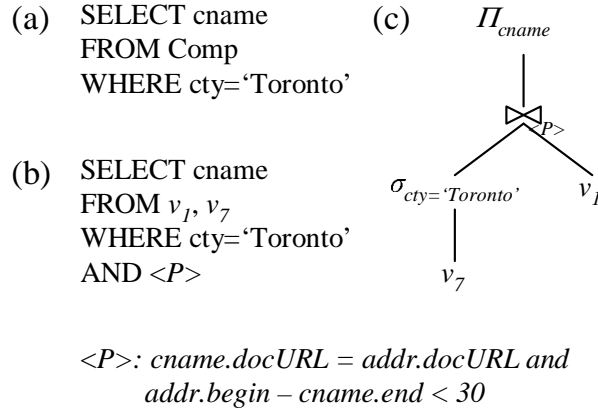
- $v_6(\text{Emp}, \{\text{hiredate}\}, C_2, E_4)$
- $v_7(\text{Comp}, \{\text{addr}, \text{cty}, \text{cntry}\}, C_1, E_8)$

The definition of v_1 means that the attribute *cname* in T-table *Comp* can be obtained by invoking IE system E_1 on document collection C_1 . Note that some views overlap, e.g. v_3 and v_5 have the *ename* attribute in common.

One might argue that extraction views are not new, since many modern database systems already have the infrastructure to plug in external data sources, e.g. using table-valued functions. However, in systems that support such functionality, queries have to call these functions explicitly, which is equivalent to explicitly specifying the extraction view in the query, leaving no room for the optimizer to choose among multiple alternative views to answer the query. Furthermore, the currently supported table-valued functions often provide no statistics to the optimizer regarding their execution cost or output cardinality or quality, which makes it extremely hard to optimize queries that reference them. However, this infrastructure is not completely unrelated to our work. One of our goals is to leverage modern DBMS support of external data sources (e.g. table-valued functions) to enable support for extraction views, by using and extending the properties of such structures to come up with a cost model as discussed in Section 4.4.

We often need to construct the tuples of a T-table from the tuples returned by multiple extraction views. In Example 5, v_1 and v_7 together cover all the attributes of the T-table *Comp*. However, with only the extraction views, it is not possible to determine which v_1 tuple belongs to which v_7 tuple. Thus a join condition is needed to determine which values actually belong to the same *Comp* entity. The concept of joining or linking the outputs of IE systems based on positional information or other metadata already exists in some of today’s publicly available IE frameworks. For example, UIMA [15] allows the definition of *aggregate analysis engines*, which operate on the outputs of extraction modules, potentially using extraction metadata to join them. Although in many cases, it is necessary to understand the semantics of the text to determine which values belong together, this can often be determined by analyzing the position of attribute values within the documents. This is especially true for documents that are semi-structured, e.g. Web pages displaying search results or containing HTML tables. This is where the lineage information produced by the IE systems becomes useful. We exploit this information by defining *joiners*.

Figure 3.1: (a) Original query (b) Equivalent query (c) Execution plan



Definition 4. A joiner j is defined as $j(T, A, C, P)$, where:

- T is a T -table on which j is defined
- A is the set of attributes in T that j covers
- C is the document collection on which j applies
- P is a predicate (or set of predicates) on the values and/or lineage of the attributes in A .

The joiner definition is specific for a particular document collection, since the document format in one collection can be different from another.

Example 6. Continuing our running example, we can define the joiner $j_1(Comp, \{cname, addr\}, C_1, P)$ where:

$P = [cname.docURL = addr.docURL \text{ AND } addr.begin -- cname.end < 30]$

This joiner definition means that a tuple that contains a $cname$ value (e.g. returned from v_1), and a tuple that contains an $addr$ value (e.g. returned from v_7) belong to the same logical entity if the $addr$ value is located less than 30 characters after the $cname$ value within the same document.

Using this joiner, the query in Figure 3.1(a) is equivalent to the query given in Figure 3.1(b) on v_1 and v_7 using the join condition P , which would potentially result in the execution plan in Figure 3.1(c). Section 4.2 explains in more detail how such an execution plan can be built.

Note that in Example 5, some extraction views (e.g. v_4 and v_6) use the same IE system to extract two different attributes. On their own, these two views return the exact same date values. However, with the other views and with the proper joiners, these two views can contribute different information to queries.

3.2.4 Document Retrieval

Since extraction is a lengthy process, it is more efficient to skip processing documents that are not likely to produce results. It is possible to use the relations and predicates in the query to determine whether or not a document is relevant. The work in [19] introduces four strategies to select a set of relevant documents.

In our work, we support two different document retrieval strategies: *scan* and *filter-scan*. The *scan* strategy sequentially feeds all documents to the IE system. In the *filter-scan* strategy, query predicates are used to determine the relevance of a document to the query. Query predicates are used to derive keywords and search for them in the document. If a document contains one or more of the keywords, it is passed for extraction. For example, in Figure 3.1(c), the condition $[cty = 'Toronto']$ can limit the documents processed by v_7 to only those containing 'Toronto'. An alternative approach can be used for document collections that offer a keyword search interface. Since the documents in these collections are already indexed, a keyword search can be issued using the constants obtained from the filtering predicates, and only the returned documents are processed by the IE system. Note, however, that the *filter-scan* strategy can only take advantage of equality predicates. Predicates such as $[Age > 18]$ or $[Country \neq 'USA']$ cannot be used to filter documents.

Generally, document retrieval strategies can affect both the cost and the quality of extraction because the skipped documents may contain correct and/or incorrect matching tuples. Retrieval strategies can be considered as physical access paths to the extraction views. More document retrieval strategies can be integrated into the system to select a set of query-relevant documents.

3.2.5 System Timeline

Objects and components described previously in this chapter are defined and used at different points in time.

IE System Registration Time

This is the phase where IE systems are added to the framework. Registering an IE system tells our framework how to invoke the IE system and what kind of information it can extract. This updates the domain universe \mathcal{D} (Definition 1). Registering IE systems is application-independent, and can be done at any time, causing the domain universe to expand as new kinds of extractable information become available.

Application Design Time

Given a particular application or query workload, this is where T-tables, extraction views, and joiners are defined, using the available IE systems and attribute domains. The application or workload cannot be started until this phase is completed.

Query Time

Queries are posed over the defined T-tables. During query optimization, the optimizer finds the best execution plan using a combination of the defined extraction views and joiners (more details in Section 4.2). During query execution, the IE systems associated with the selected views are invoked, extracting information from the corresponding document collections and returning them as relational tuples.

Chapter 4

Query Optimization

In this chapter, we discuss the current work done to perform optimization for on-demand information extraction tasks. In the JITE approach, we utilize the different framework components described in Chapter 3. We first start by defining our optimization objective in Section 4.1. Section 4.2 describes the process of enumerating different query plans, and the algorithm used to select the best query plan. We describe further optimization opportunities in Section 4.3, and describe how to estimate the parameters and collect statistics about cost and quality of extraction in Section 4.4.

4.1 Optimizing IE tasks

IE algorithms vary in complexity, ranging from simple pattern matching to complex text analysis (e.g., natural language processing). This variation causes IE systems to have different processing costs. In queries that have extraction views, the cost of processing documents using their IE systems is the main contributor to the overall execution cost. We define the cost of an extraction view v , $Cost(v)$, as the time taken to perform the extraction on its associated document collection. In Section 4.4, we show how to estimate this cost. We define the efficiency, Eff , as the inverse of the $Cost$.

$$Eff(v) = \frac{1}{Cost(v)} \quad (4.1)$$

The variation in IE algorithms also leads to different kinds of extraction errors. Examples of these errors include producing erroneous tuples, encountering different forms of the same entity and not being able to recognize that fact, or missing some valid tuples. In order to correct these errors, the extracted data may need to be cleaned as part of the query execution. Data cleaning can include reference reconciliation techniques (also known as record linkage, among other names [14, 28]). These extraction errors contribute to the quality of extracted data. For example, one IE system might generate many false positives, while another IE system might be more conservative, at the expense of possibly missing correct tuples (false negatives). We use the same quality metric proposed in [19], which combines both *precision* and *recall* of the extracted data¹. The quality of a view v is:

$$Q(v) = \sqrt{Pc(v) \times Rc(v)} \quad (4.2)$$

where $Pc(v)$ and $Rc(v)$ are the precision and recall of the output relation produced by v , respectively. The statistical estimation of Pc and Rc is discussed in Section 4.4.1. The definition of Q assumes equal weight (importance) for the precision and recall, but it can be modified to favor either of them.

There is a natural trade-off between the output quality and the query execution time. Simple extraction techniques are fast, but may erroneously output invalid entities or relations. More advanced methods may take longer to execute, but produce better results. Although some prior research efforts [30, 31] focused only on optimizing execution cost, others [20, 21, 23] took both factors into account while optimizing a query. In our work, we follow the latter path.

The balance between efficiency and quality is user-specific. One user may be interested in a “quick and dirty” answer, while another user may be interested in a high quality answer regardless of how long it would take. We adopt the user-specified parameter approach proposed in [19] to reflect the user preference between cost and quality. A parameter, denoted by w ,

¹We acknowledge that other measures, such as the geometric mean (G-Score), exist to combine both precision and recall in IR systems.

represents the balance between efficiency and quality when comparing different query plans. Using this parameter, we can combine both efficiency and quality in one *goodness* metric, as proposed in [19]. The goodness of an extraction view v is defined as:

$$G(v, w) = \text{Eff}(v)^w \times Q(v)^{1-w} \quad 0 \leq w \leq 1 \quad (4.3)$$

Our objective is to maximize the goodness, given the user parameter w . During optimization, the query optimizer estimates the goodness score for alternative query plans, and chooses the one with the highest goodness. For a query plan P that contains multiple views, the goodness of P is:

$$G(P, w) = \text{Eff}(P)^w \times Q(P)^{1-w} \quad 0 \leq w \leq 1 \quad (4.4)$$

Estimating the efficiency of P depends on the plan cost, which is estimated by the DBMS. We describe the quality estimation of plans in Section 4.4.1.

4.2 Plan Enumeration and Selection

The problem of choosing which views and joiners to use for query evaluation is not trivial, given that we allow overlapping views, and that some views may not be joinable with each other due to the lack of appropriate joiners. The efficiency and quality constraints imposed by the user also make it harder to find the best combination of views and joiners to generate the required output. Traditional view matching, as that used for materialized views, is not adequate, because of its reliance on the concept of subsumption, as described in Section 2.3.2. For a T-table to be replaced by a view, the view must produce at least all the attributes and tuples required from that T-table. However, it is often the case that there is no single view that covers all the required attributes, but rather multiple extraction views that have to be joined. Traditional view matching only considers joining views if all of them include the primary key of the base table they are defined on, and does not consider joining views based on user-defined join conditions.

In addition, failed matching is not a problem in the case of materialized views, since a failed match simply means that the query block has to be evaluated using the base tables. However, in our case, a failed match means that the query can not be evaluated at all, since the extraction views *are* the source of the data.

Example 7. Given the T-tables defined in Example 4, consider the following query:

```
SELECT ename FROM Comp, Emp
WHERE cname = ecomp
AND job = 'Manager'
AND cty = 'Toronto'
```

Given the extraction views in Example 5 and the joiner in Example 6, we can see that v_5 covers the attributes and tuples from the *Emp* T-table. Extraction view matching needs to recognize that v_1 and v_7 can be used together (using the joiner j_1) to produce the required attributes from the *Comp* T-table.

In our system, we select the plan with the highest goodness, which is an aggregation of the plan execution cost and the plan quality (Section 4.1). We view the process of generating an optimal plan as a multi-objective optimization, where the objectives are the quality and the cost of the plan. For two possible plans P_1 and P_2 of a given query, P_1 *dominates* P_2 if both the quality and the efficiency of P_1 are greater than the quality and the efficiency of P_2 , respectively. The goodness metric in Equation 4.3 is monotone w.r.t. the quality and the efficiency of plans. Therefore, it is possible to prune plans that are dominated w.r.t both objectives. The same pruning criterion is used while generating sub-plans. The reason is that the efficiency of a plan is monotone w.r.t. the efficiencies of the sub-plans, and similarly, the quality of a plan is monotone w.r.t. the qualities of the sub-plans (see Section 4.4.1). It follows that replacing a sub-plan P'_1 with another sub-plan P'_2 that dominates P'_1 results in increasing both the efficiency and the quality, and hence the goodness, of the parent plan.

Algorithm 1 describes our plan enumeration and selection module. The *getCandidatePlans* function takes the following as input: the set A of all required attributes (referenced in the SELECT and WHERE clauses of the query), the set β of all the query predicates, and w the user-defined weight preference between efficiency and quality. The function returns a candidate set of execution plans that are not dominated. We obtain the optimal plan by computing the goodness of each candidate plan, and returning the plan with the maximum goodness. This algorithm has been developed by Amr El-Helw [11].

Algorithm 1 finds the candidate execution plans for a given query by recursively finding candidate plans of sub-expressions of the query that are not dominated, and enumerating all

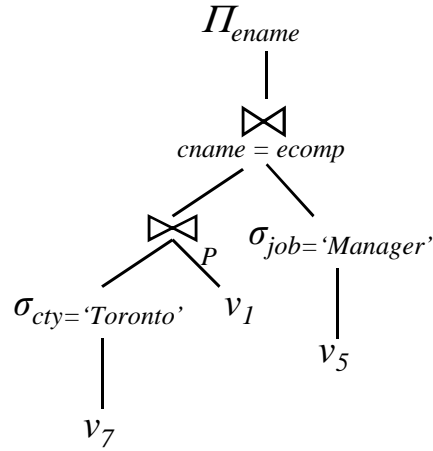


Figure 4.1: Execution plan for query in Example 7

possible physical implementation of the current operation (e.g., selection, projection or join). The algorithm makes use of an in-memory structure (*MEMO*), which saves the candidate plans found for every set of attributes and predicates that has been previously encountered in order to avoid repeated work. Each enumerated plan is evaluated, using the function *MemoAdd*, in order to determine whether or not it is dominated by, or dominates, other plans. Dominated plans are pruned. After enumerating all plans, the set of non-dominated plans are returned.

Splitting the original query into smaller expressions is performed based on the required attributes (lines 12-14). For each subset of the given attributes and the corresponding predicates the algorithm finds the candidate plans to obtain these attributes. These plans could either involve using one extraction view (if there exists a view that produces all desired attributes), or multiple joined views. In the latter case, these views must be joinable using a join predicate in the query itself or a joiner.

Given two sets of candidate sub-plans for two sub-expressions, finding the candidate plans that join these two sub-expressions is achieved using the *getCandidateJoinPlans* function (line 18). This function enumerates different join methods and orders, and returns the set of non-dominated plans. This join plan either uses join predicates from the query, or pre-defined joiners.

For the query in Example 7, the algorithm is called with $A = \{ename, ecomp, job, cname, cty\}$ and β containing the three predicates. After trying different combinations, it would find that

$(ename, ecomp, job)$ can be obtained from v_5 , $cname$ can be obtained from v_1 , and cty can be obtained from v_7 . The views v_1 and v_5 can be joined using the join predicate in the query, while v_1 and v_7 can be joined using the joiner j_1 . A possible plan to execute the query is shown in Figure 4.1.

4.3 Join Predicate Push-down

We describe a further optimization opportunity to reduce the number of documents to process in a query with two or more extraction views. Similar to the basic principle of semi-join algorithm in distributed joins [35], the join predicates are pushed down the extraction view node to guide the extraction. This reduces the extraction cost, as opposed to data transmission cost in distributed databases. The pushed down predicates introduce additional filtering, regardless of which document retrieval strategy is used.

The idea behind pushing down the join predicate is to avoid processing a particular document if it is possible to infer that the outputs from that document will not contribute to the result of the query. For instance, in Figure 3.1(c), the join condition specifies that tuples can be joined only if they come from the same document. If the IE system corresponding to v_7 is invoked first on each document, and the filtering predicate is applied, the processing of v_1 can be limited to documents whose $docURL$ values are in the filtered outputs of v_7 , instead of the whole document collection. If the query includes filtering predicates on both v_7 and v_1 , then it would also be possible to evaluate v_1 first and use its outputs to guide the evaluation of v_7 . Choosing which alternative to use depends on the estimated selectivity of each filtering predicate.

In our implementation, we use a heuristic in which the optimizer forcefully pushes the join predicates down to the extraction views only when the joined attributes are expected to come from the same document. This optimization leads to processing fewer documents and has no effect on output quality, since the discarded documents do not contribute to the results.

4.3.1 Extraction Output Caching

Caching extraction results to avoid reprocessing documents improves the performance and reduces the amount of extraction performed during query processing. For example, without caching, a nested loop join between two extraction views would repeatedly process all documents of the inner child extraction view, for each tuple in the outer result set. We store the outputs of running extraction views over text documents, and retrieve the cached results when the underlying corpus did not experience significant changes. Currently, we keep the cache for each query, and retrieve it whenever a cached extraction view is requested on a pre-processed file. Following queries refresh the cache and re-extract from their files. It is possible to apply a more sophisticated caching by storing the intermediate and final query results with corresponding query predicates, and use them in following queries (similar to materialized views described in Section 2.3.2). We leave advanced caching techniques for future work.

4.4 Cost Estimation

Cost estimation relies on statistics maintained by the system. Table 4.1 gives the statistics needed to estimate the cost and quality of information extraction. We explain in Section 4.4.2 how these statistics can be obtained.

Note that for some views, the statistics $Time_{doc}$ and $|Rows_{doc}|$ may be available, while for others, $Time_{KB}$ and $|Rows_{KB}|$ may be available. Choosing which to collect depends on the nature of the associated IE system, and the kind of information it extracts from text documents. For example, consider a document collection that represents news articles. An IE system that extracts mentions of places definitely depends on the document size, and would benefit from having the $Time_{KB}$ and $|Rows_{KB}|$ information available. In contrast, an IE system that extracts the author of each article does not depend on how short or long the article is, and would spend roughly the same amount of time on all documents. Hence it makes more sense to maintain $Time_{doc}$ and $|Rows_{doc}|$ for it.

Given the statistics in Table 4.1, a simple cost estimate of using a view v , with its associated IE system, is as follows:

Table 4.1: Database statistics

Term	Meaning
$ Docs(C, x) $	Average number of documents in document collection C returned by document retrieval strategy x
$DocSize(C)$	Average size of documents in C (measured in KB)
$ Rows_{doc}(v) $	Average no. of tuples extracted from a single document by the IE system associated with v
$Time_{doc}(v)$	Average time to retrieve and process a document by the IE system associated with v
$ Rows_{KB}(v) $	Average no. of tuples extracted by the IE system associated with v , per KB of text
$Time_{KB}(v)$	Average processing time by the IE system associated with v per KB of text
$Pc(v, x)$	Approximate precision of running v over its document collection using retrieval strategy x
$Rc(v, x)$	Approximate recall of running v over its document collection using retrieval strategy x

$$Cost(v, x) = \begin{cases} |Docs(C_v, x)| * Time_{doc}(v) \\ |Docs(C_v, x)| * DocSize(C_v) * Time_{KB}(v) \end{cases} \quad (4.5)$$

where C_v is the document collection mapped to v . The choice of which cost formula to use depends on the availability of either $Time_{doc}(v)$ or $Time_{KB}(v)$. Similarly, the output cardinality of v can be estimated as:

$$|Rows(v)| = \begin{cases} |Docs(C_v, x)| * |Rows_{doc}(v)| \\ |Docs(C_v, x)| * DocSize(C_v) * |Rows_{KB}(v)| \end{cases} \quad (4.6)$$

The design and collection of previous statistics metrics are implemented by Amr El-Helw, and utilized in the view matching algorithm, Algorithm 1.

4.4.1 Quality Estimation and Propagation

Unlike execution cost, current DBMSs do not have a well defined way to propagate quality along the query plan. We adopt the propagation model described in [19], where extraction errors are assumed to be independent of any predicate selectivity. The quality of a view v reflects the quality of its corresponding IE system E . This quality is estimated using Equation 4.2 with the precision and recall values defined in Table 4.1. Consider a given intermediate node P in the query plan. Let P_1, \dots, P_k be the child nodes of P . It can be easily shown that the precision and recall at node P can be estimated as follows:

$$Pc(P) = \prod_{i=1}^k Pc(P_i) \quad (4.7)$$

$$Rc(P) = \prod_{i=1}^k Rc(P_i) \quad (4.8)$$

However, empirical studies in [19] showed that this assumption underestimates the actual precision. Therefore, the authors introduced a boosting mechanism to calculate the precision of joined views as:

$$\bar{P}c(P) = \prod_{i=1}^n \frac{Pc(v_i)}{1 + Pc(v_i)} \quad n \geq 2 \quad (4.9)$$

4.4.2 Statistics Collection

In addition to the traditional database statistics, the information in Table 4.1 is required to optimize queries on text documents. The statistics also need to be kept up-to-date to avoid large estimation errors that result from outdated statistics. Statistics collection in databases is usually an offline process that is performed separately from query processing. Statistics are not updated whenever there is an update to the data for obvious cost considerations. Traditional systems try to address this problem by periodically updating the stored statistics. It is also possible to perform some on-the-fly updates for the database statistics during query processing, only for the statistics required by a given query [12].

Statistics collection tools often use random sampling to reduce the time spent to collect these statistics. We use sampling to compute the statistics in Table 4.1. For a document collection

C , the value $|Docs(C)|$ is usually available and easily accessible. For *scan* retrieval strategy, $|Docs(C, scan)|$ is the actual $|Docs(C)|$. However, $|Docs(C, filter - scan)|$ must be estimated during query time depending on selectivity of the query predicates. For the remaining statistics, a random sample of size s can be taken from the complete document collection and used for statistics collection. Assuming the documents in this sample are d_1, d_2, \dots, d_s , the average document size $DocSize(C)$ can be computed as follows:

$$DocSize(C) = \frac{1}{s} \sum_{i=1}^s size(d_i) \quad (4.10)$$

For a view v , its associated IE system is invoked on the document sample. Let $Time_s(v)$ be the time it takes to process all the documents in the sample, and let $Rows_s(v)$ be the set of rows extracted from these documents. The statistics can be computed as follows:

$$|Rows_{doc}(v)| = |Rows_s(v)|/s \quad (4.11)$$

$$Time_{doc}(v) = Time_s(v)/s \quad (4.12)$$

$$|Rows_{KB}(v)| = |Rows_s(v)| / \sum_{i=1}^s size(d_i) \quad (4.13)$$

$$Time_{KB}(v) = Time_s(v) / \sum_{i=1}^s size(d_i) \quad (4.14)$$

As explained earlier in Section 4.4.1, the precision and recall are more challenging to collect. To estimate the precision and recall of an extraction view v , the sample obtained from the corresponding document collection C needs to be annotated (possibly by an expert). Annotating the sample documents tells us the tuples that should be ideally extracted from these documents. Let $Ideal_s(v)$ refer to this set of rows. The precision and recall can be computed as follows:

$$Pc(v, x) = \frac{|Rows_s(v) \cap Ideal_s(v)|}{|Rows_s(v)|} \quad (4.15)$$

$$Rc(v, x) = \frac{|Rows_s(v) \cap Ideal_s(v)|}{|Ideal_s(v)|} \quad (4.16)$$

As in traditional DBMSs, in the absence of statistics on tables, the system assigns default values for quality.

Algorithm 1 $\text{getCandidatePlans}(A, \beta, w)$

```
1:  $\mathcal{P} \leftarrow \text{MemoLookup}(A, \beta)$ 
2: if  $\mathcal{P} \neq \phi$  then
3:   return  $\mathcal{P}$ 
4: end if
5:  $V \leftarrow \text{getAllExtractionViews}()$ 
6: for each  $v \in V$  do
7:   if  $(A \subseteq \text{attributes}(v))$  then
8:      $P \leftarrow \text{plan}(\sigma_\beta(v))$ 
9:      $\text{MemoAdd}(A, \beta, P)$ 
10:  end if
11: end for
12: for each  $X \subset A$  do
13:    $Y \leftarrow A - X$ 
14:    $(\beta_X, \beta_Y, \beta_{\bowtie}) \leftarrow \text{split}(\beta, X, Y)$ 
15:    $\mathcal{P}_X \leftarrow \text{getCandidatePlans}(X, \beta_X, w)$ 
16:    $\mathcal{P}_Y \leftarrow \text{getCandidatePlans}(Y, \beta_Y, w)$ 
17:   if  $(\mathcal{P}_X \neq \phi \text{ AND } \mathcal{P}_Y \neq \phi)$  then
18:      $\mathcal{P} \leftarrow \text{getCandidateJoinPlans}(\mathcal{P}_X, \mathcal{P}_Y, \beta_{\bowtie}, w)$ 
19:     for each  $P \in \mathcal{P}$  do
20:        $\text{MemoAdd}(A, \beta, P)$ 
21:     end for
22:   end if
23: end for
24:  $\text{Candidates} \leftarrow \text{MemoLookup}(A, \beta)$ 
25: return  $\text{Candidates}$ 
```

MemoAdd(A, β, P)

26: $\mathcal{P} \leftarrow MemoLookup(A, \beta)$

27: **if** P is not dominated by any plan in \mathcal{P} **then**

28: add P to the MEMO structure

29: **end if**

30: **for** each $P' \in \mathcal{P}$ **do**

31: **if** P dominates P' **then**

32: remove P' from MEMO

33: **end if**

34: **end for**

Chapter 5

Implementation and Technical Details

This Chapter describes the details of system implementation. It explains how the components of the framework mentioned in Section 3.2 are implemented. We integrated our framework inside an existing RDBMS. In our prototype, we used Apache Derby [1] version 10.6.1 to support JITE. Derby is an open source relational database management system implemented entirely in Java and available under the Apache License, Version 2.0. The following sections describe the modifications and additions done in Derby to support JITE.

5.1 Extending an RDBMS to Support Information Extraction

As explained in Chapter 1, JITE can be implemented as light-weight modifications to existing RDBMS. We utilized existing components in Derby to avoid fundamental changes to the internals of the DBMS.

5.1.1 Supporting Information Extraction Systems

Different information extraction systems have different implementations and are not usually designed for reusability. Section 2.1 described two frameworks that standardize the development of IE systems. In our system, we support extractors developed using Apache UIMA [15]. We

implemented a wrapper to use UIMA analysis engines to perform information extraction. The extractors are developed in Java, but their code can use libraries developed in other languages. For instance, we developed an analysis engine that runs GATE [10] applications, and two other analysis engines to wrap generic named entity recognition (NER) libraries: LingPipe¹ and Balie².

Supporting Apache UIMA has a few advantages. First, UIMA is implemented in Java, which allows close integration between the UIMA framework and Derby. Also, it supports analysis engines to be plugged in the system and used directly without the need for any configurations. This allows the users to use any UIMA extractors as black boxes without prior knowledge about their configurations or about how they work. And since Apache UIMA is a standardized framework, developing applications for it should be an easy process to learn. In our future work, we plan to implement a direct integration between Derby and GATE, without the need for the GATE-to-UIMA wrapper.

5.1.2 Implementing the Data Model

In this section, we will explain how the JITE data model is represented in Derby, and how it is affected by different operations, such as adding a new extractor, or running statistics.

The components of the data model can be considered as metadata for each database instance. Information about the T-tables, extraction views, attributes and their corresponding mapping and lineage, and different performance and quality statistics are stored in the system catalog tables of each database instance. We will describe the development of a database instance through an example. All of the operations are performed through custom SQL commands that we added to Derby.

Data Sources

Data sources represent repositories that contain unstructured data. A source can include a set of text documents, or it can be a website, or theoretically any set of unstructured documents.

¹<http://www.alias-i.com/lingpipe/>

²<http://balie.sourceforge.net/>

Currently, only corpora containing text documents are supported, and we leave implementing other data source types for future work. Implementing an interface for a new source type would require implementing some functionalities such as documents retrieval, keyword search, and documents access (retrieve data of a certain document).

Adding a data source is done by a custom SQL statement. A sample source that contains documents about company information is:

```
CREATE SOURCE data_source FROM '/Company Data/';
```

where the address is an absolute path, in the case of file system data source. An HTTP data source would replace the path string with the URL of the website.

Information Extraction Systems (Extraction Views)

Users start by registering IE systems to the system. This registration makes the system aware of such extractors. Currently, we only support Apache UIMA analysis engines. An analysis engine has an XML descriptor file that includes all of the extractor's properties: input, output, classes, etc. An extraction view reflects a UIMA extractor. Derby supports user functions that returns tables (table-valued functions). An implementation of table functions is its Virtual Table Interface (VTI). A VTI can be created using an SQL command, and the definition should include the language of its implementation, the parameters, class name, and function to run. The function to run should return an object of the type `ResultSet`, that would be used as a source for structured data. We utilize the VTI object in Derby to implement the extraction views, with some modifications. We added a custom VTI creation statement that meets our needs as extraction views. It wraps the original VTI creation statement, adding some default values such as UIMA wrapper class and parameters. A sample SQL command to create a VTI that reflects an IE system extracting employees and companies they work in is as follows:

```
CREATE VTI E_Comp
RETURNS TABLE
(
    emp_name VARCHAR(50),
    comp_name VARCHAR(100)
```

```

)
FROM DESC '/Extractors/EmployeeCompanyDescriptor.xml'
(
    'COM.EComp.EmployeeName',
    'COM.EComp.CompanyName'
)
SOURCE data_source;

```

The previous SQL statement creates a few entries in the database instance catalog tables. First, it creates a VTI in the `SYS.SYSVTIS` table that holds information about the available VTIs. Information about this VTI is its name (`E_Comp`), signature, and the corresponding UIMA analysis engine descriptor file (`DESC`). It assigns this VTI to run on a specific (`data_source`). The reason why VTIs (or extraction views) are linked to a certain data source is that an extractor – usually – works on a specific type of data. For instance, an extractor may work on XML documents, but cannot perform extraction on plain text documents. The `TABLE` attributes returned by the VTI represent the (selected) possible output of the extractor, and the class types are the UIMA types that correspond to these attributes. Information about the VTI columns are stored in the `SYS.VTICOLUMNS` catalog table. It is the responsibility of the user to obtain and assign information about the output of different extractors, and what data they operate on, as we consider IE systems as black boxes. Adding a new extraction view (VTI) expands domain universe by adding new attributes and a way to extract them, or provides a new way of extracting existing attributes.

T-Tables

From a data integration point of view, extraction view (VTIs) represent the local data sources, and the T-tables correspond to elements of the global schema. The attributes of a T-table are selected from the available attributes domain. Supporting a new table type in Derby requires implementing some classes, and modifying the system to realize the new table type. Performing most operations in Derby is done using SQL statements. The SQL statements are compiled into query trees that get executed to reflect the corresponding command. Creating a table is no

different. We added a new table creation node. We modified the SQL grammar to support creating a T-table. A sample SQL statement to add a new T-table is:

```
CREATE TEXTTABLE EMPLOYEE
(
    EmployeeName VARCHAR(50),
    CompanyName VARCHAR(100)
);
```

The new T-table creation node is added to the query tree. The execution of a T-table creating node can be considered as a regular table, but T-tables do not support data insertion, deletion, or update. Also, currently, we do not support alteration of the table definition. The catalog tables are changed to apply the new changes; adding the new T-table to the list of tables (with an appropriate table type), and the columns with their properties to the system columns (in `SYS.SYSCOLUMNS` table), referencing the T-table columns. The mapping (reference) represents a view coverage over T-tables. The mapping between the T-tables and extraction views is currently done manually. The command to map attributes of an extraction view to a T-table is as follows:

```
ALTER VTI E_Comp REFERENCES Employee(EmployeeName, CompanyName);
```

The previous SQL statement updates the entries `emp_name` and `comp_name` in `SYS.VTICOLUMNS` table to reference the `EmployeeName` and `CompanyName` column from the `SYS.SYSCOLUMNS` table.

Joiners

Joiners are defined over attributes of a T-table. This allows any views sharing these attributes to use the same joiner. They are defined using the SQL statement:

```
CREATE JOINER e_emp_comp ON Employee (EmployeeName, CompanyName)
AS '#EmployeeName#_docid = #CompanyName#_docid
AND #CompanyName#_begin - #EmployeeName#_begin < 20';
```

This previous joiner can include any user-defined join condition in the `AS` clause. When the query on a T-table is rewritten in terms of VTIs and the joiner is used, the joining condition of

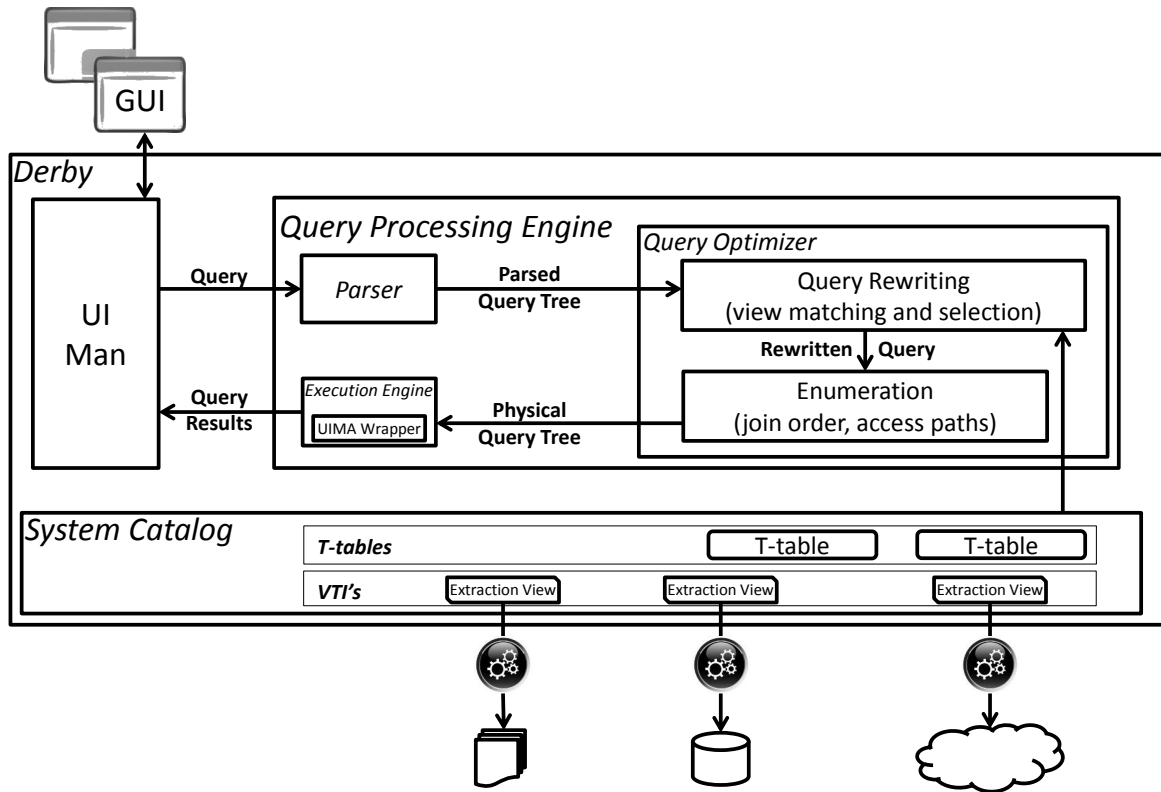


Figure 5.1: System Architecture

the joiner is added to the SQL statement to join the VTIs.

5.1.3 System Architecture

Apache Derby includes the common modules found in most DBMS's nowadays. Figure 5.1 shows the system architecture of Derby, along with the components that are added to support JITE.

User Interface Manager

The user interface manager (UI Man) is responsible for all communication with the user: DB connection, updating catalog, querying, and displaying the results. The default UI tool in Derby is called `ij`; a command-line interface to run SQL commands and statements on a database and display queries results. We developed a more user-friendly graphical user interface (GUI) for Derby, mainly to support different operations performed during the JITE process. The GUI communicates with the UI Manager to pass commands and results. The UI Manager transforms the user actions to appropriate commands (most probably SQL queries) sent to the query processing engine of the DBMS.

Query Processing Engine

The query processing engine is the core of Derby. It includes components to parse, optimize, and execute SQL statements. The parser takes plain text SQL statement, and parses it using a defined grammar.

The parser is generated using JavaCC from the grammar file (`SQLGRAMMAR.JJ`). The parsing process generates a parsed query tree that represents the SQL statement in hand. Every node in the tree represents a database operation. Assume we have the following T-table:

Employee(EmployeeName, CompanyName)

with two extraction views defined on it: *E_emp(emp_name)* and *E_comp(comp_name)*, and the joiner *e_emp_comp* defined above. Consider the following SQL statement:

```
Select EmployeeName, CompanyName
FROM Employee;
```

This is a single operation SQL statement to select two columns from a T-table. The parsed query tree is then passed to the query optimizer.

The query optimization phase consists of three steps: preprocessing, optimization, and modifying access paths. Briefly, the preprocessing performs rewritings in predicates, and it flattens subqueries. In addition, we added our own view matching algorithm here to transform

queries on the T-tables to equivalent queries on the extraction views. We implemented a version of Algorithm 1 as a preprocessing phase, and heuristically, we prune subtrees with lower goodness estimates to limit the expansion of plans space. Continuing the previous example, the parsed query plan initially references the T-table `Employee`. The view matching algorithm in the preprocessing phase finds a plan to use the two extraction views, `E_emp` and `E_comp`, and join them using `e_emp_comp`. The SQL query is rewritten to:

```
Select emp_name, comp_name
FROM E_emp, E_comp
WHERE emp_name_docid = comp_name_docid
AND comp_name_begin - emp_name_begin < 20;
```

Other relational objects in the query tree are optimized normally. The Derby optimizer then continues to the optimization step, which enumerates all possible join permutations and algorithms, and selects the best join order. Modifying access paths pushes predicates down after the best access path is determined for each component. Optimization generates an execution query tree, with the physical operator nodes for required operations.

The execution engine takes an optimizer query tree that includes extraction views (VTIs) nodes instead of the T-tables nodes. The execution engine runs on the root of the query tree node. Each node calls its children according to its type, or performs an execution if it is a leaf node. A nested-loop node, for instance, would open the left node, get a tuple from the left, open the right node, get all tuples from the right, compare the left tuple with each of the right tuples based on its join criteria, and return the tuples that satisfy the join conditions. Then, it gets another tuple from the left, and so on.

As for extraction views (VTIs) nodes, our implementation target streams the extraction results to the user. We use the `openCore` function to set the parameters of execution: use the recommended document retrieval strategy to initialize the list of extraction files and initialize the extractor. We developed a wrapper to UIMA framework to initialize analysis engines and perform extraction. When the execution starts, we take a file from the files list and perform extraction on it (again, using the UIMA wrapper). This should return a `ResultSet` object that includes the extraction results. Rows of the `ResultSet` object are returned one by one, until they are finished. The next call triggers extraction from the next file in the list, if it exists, or

returns a `NULL` , marking the end of extraction for this node. The results are streamed, tuple by tuple, to the UI Manager, which displays them appropriately on the GUI.

Chapter 6

Experiments, Results and Evaluation

We performed a few experiments to study just-in-time extraction (JITE) benefits and performance. We divide the experiments into two main categories. The first category contains experiments that compares our system – applying the JITE approach – to other approaches and systems. The second category tests different optimizations, benefits and performance issues.

6.1 Experimental Settings

We start by describing the different settings that we used to perform experiments.

Computing Environment

All our experiments were conducted on a SunFire X4100 server (see Table 6.1). We implemented our prototype in Java inside the Apache Derby DBMS [1], as explained in Chapter 5.

Available Information Extraction Systems

We extended Derby by adding support to the UIMA [15] framework, which allows us to plug in IE systems (known as *analysis engines* in UIMA terminology). For these experiments, we

Table 6.1: Computing environment for the experiments

CPU	Two dual-core AMD Opteron 280 CPUs
RAM	8 GB
Disk	Two 72 GB disks, in RAID-0 configuration
Operating System	OpenSuSE Linux 10.1
Runtime Environment	Java 1.6.0_18 (Sun)

Table 6.2: IE systems

IE system	Domains(s)
E_N	name
E_C	organization
E_J	position
E_M	email
E_{NC}	name, organization
E_{NJ}	name, position
E_{CJ}	organization, position
E_{NCJ}	name, organization, position
E_{NCM}	name, organization, email
E_L	location
E_{CL}	company, location

built IE systems using the GATE [10] framework, UIMA analysis engines, and the named entity recognizers LingPipe¹ and Balie², all of which can be made to produce output that can be recognized by UIMA through appropriate wrappers. Using these frameworks and components, we built the IE systems shown in Table 6.2.

¹<http://www.alias-i.com/lingpipe/>

²<http://balie.sourceforge.net/>

Table 6.3: Extraction views

View	T-Table	Attribute(s)	IE system
v_E	<i>Employee</i>	<i>Ename</i>	E_N
v_{C1}	<i>Employee</i>	<i>Ecomp</i>	E_C
v_J	<i>Employee</i>	<i>Job</i>	E_J
v_M	<i>Employee</i>	<i>Email</i>	E_M
v_{ECJ}	<i>Employee</i>	<i>Ename, Ecomp, Job</i>	E_{NCJ}
v_{ECM}	<i>Employee</i>	<i>Ename, Ecomp, Email</i>	E_{NCM}
v_{C2}	<i>Company</i>	<i>Cname</i>	E_C
v_L	<i>Company</i>	<i>Loc</i>	E_L
v_{CL}	<i>Company</i>	<i>Cname, Loc</i>	E_{CL}

Data Set and Relations

We use a subset of the North American News Text Corpus³, with articles from The New York Times. We used a data set of 2,782 documents representing one month of news articles. For our experiments, we defined two T-tables:

Company(Cname, Loc) and

Employee(Ename, Ecomp, Job, Email).

Tables 6.3 and 6.4 show some of the defined extraction views and joiners respectively (all of them reference the same document collection).

Queries

For workload-related experiments, we use 50 SQL queries over *Company* and *Employee* defined manually to cover an interesting mix of selections and projections, including selections with many and with few or no database matches.

³<http://www ldc.upenn.edu>

Table 6.4: Joiners

Joiner	Attribute(s)	Predicates
j_1	<i>Ename, Ecomp</i>	<i>Ename.docURL=Ecomp.docURL and Ecomp.begin-Ename.begin <20</i>
j_2	<i>Ename, Job</i>	<i>Ename.docURL=Job.docURL and Job.begin-Ename.begin <20</i>
j_3	<i>Cname, Loc</i>	<i>Cname.docURL=Loc.docURL and Loc.begin-Cname.begin <20</i>
j_4	<i>Ecomp, Email</i>	<i>Ecomp.docURL=Email.docURL and Email.begin-Ecomp.begin <20</i>

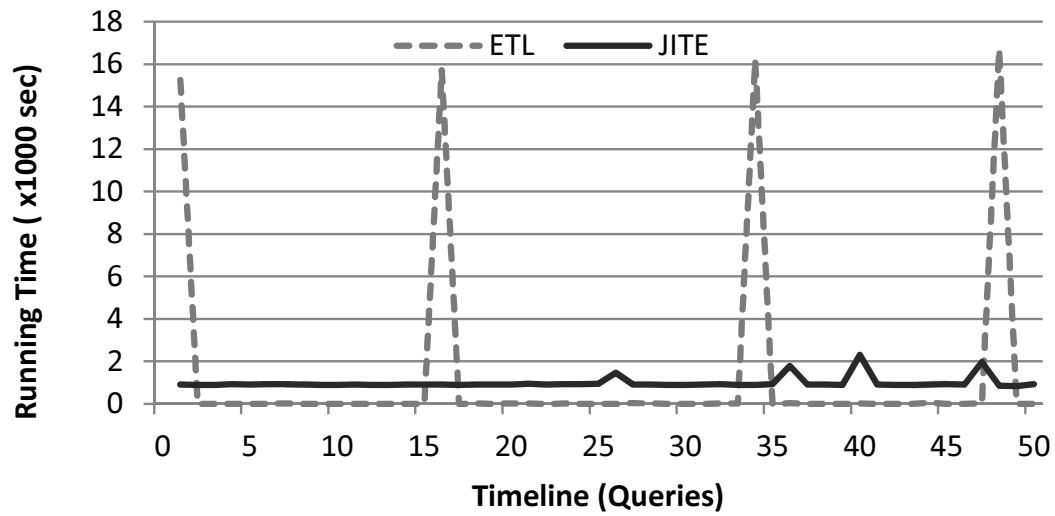
6.2 Comparing to Other Approaches

We start by comparing our system to the standard ETL approach, and then compare it to a similar system, SQoUT.

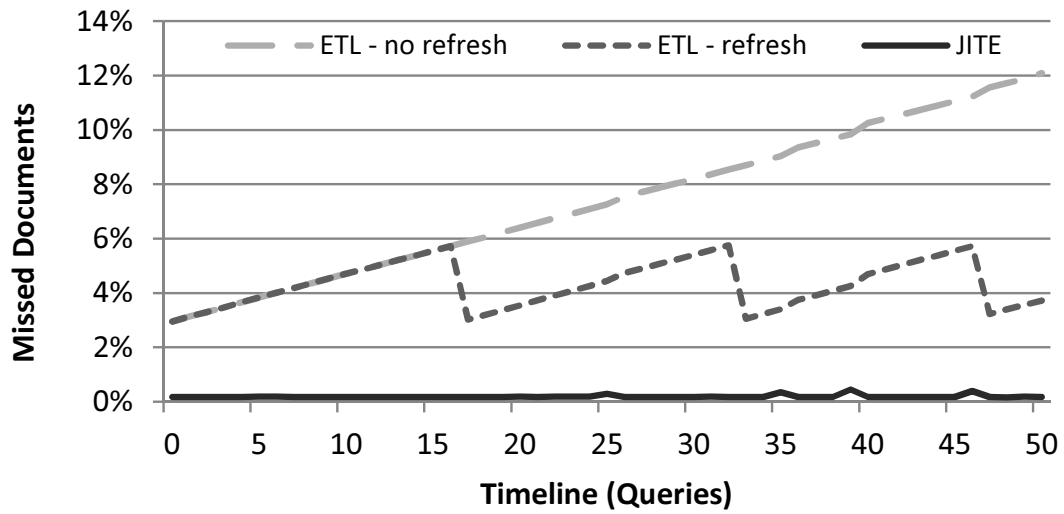
6.2.1 Comparing to the ETL Approach

We compare our Just-in-Time Extraction (JITE) approach to the baseline Extract-Transform-Load (ETL) approach. In the ETL approach, there is an offline phase where all the data is extracted using all available extractors (enough to produce all relations) and stored in a data warehouse, and then the queries are executed on that warehouse. To cope with changing data, the warehouse has to be refreshed by removing old data and re-invoking the extraction phase. In order to compare the two approaches, we add the refresh cost to the cost of the first query executed after the refresh.

Figure 6.1(a) shows the running time of each query for both approaches. The cost using JITE is almost constant in our workload. The cost of ETL starts high because of the initial loading cost, then drops significantly since subsequent queries run on the extracted data. The ETL cost peaks at refresh points. The refresh frequency determines the overall ETL cost. This refresh frequency also affects the recency of the extracted data when there are new documents being added all the time, which is the case for news articles or email archives. Since our document corpus is based on the New York Times, we use the rate by which new articles are added to the New York Times



(a)



(b)

Figure 6.1: ETL vs. JITE

website⁴, which was one new document every 3-4 minutes on average. Figure 6.1(b) shows the number of documents that are missed for every query. Using the ETL approach, a query misses all the documents that have been added since the beginning of the most recent refresh, while using JITE, a query only misses the documents added after that query starts executing.

In the extreme case, if the warehouse in the ETL approach is never refreshed after the initial load, the number of missed documents keeps increasing. With periodical refreshes, the number of missed documents increases with each query until the next refresh, causing the number of missed documents for the first query after that to drop. The refresh frequency used in our experiment caused the ETL approach to miss up to 6% of the documents. This can be lowered using a higher refresh frequency, at the expense of increasing the overall execution cost (time taken to finish extraction). Even if we decide to refresh the warehouse before each query, JITE would still miss fewer documents than ETL, since the ETL approach would miss all the documents that are added during refresh and query execution. The trade-off between data recency and execution cost is ultimately application-specific.

Note that our experiments have only two T-tables. In settings where there are more T-tables in the database, a single refresh needs to update all these T-tables, which would take more time, causing more documents to be missed, and increasing the refresh cost in Figure 6.1(a).

6.2.2 Comparing to SQoUT

We compare our approach to SQoUT [19]. In SQoUT, structured queries are posed on tables that are populated during query execution by IE systems. The two systems work in different ways and may not be comparable strictly from a performance point of view. However, SQoUT has the limitations described in Section 2.2, and can be seen as a special case of our approach. To realize SQoUT in our system, we partition the attributes of a T-table as SQoUT allows (see Section 2.2), and define extraction views such that each view returns exactly the attributes in any given partition (including the primary key). Multiple views can be defined on the same partition, using different IE systems. For this experiment, we assume all views on the *Employee* T-table are available. SQoUT can only support views $\{v_E, v_{ECJ}\}$ since $\{v_{C1}, v_M, v_J\}$ do not extract the primary key, and $\{v_{ECM}, v_{EC}\}$ share *Ecomp* with v_{ECJ} .

⁴<http://www.nytimes.com>

Table 6.5: Possible query plans under SQoUT and our system

Query	SQoUT	Our System
SELECT Ename, Email FROM Employee FROM Employee	N/A	$P_1=v_{ECJ} \bowtie_{j_4} v_M$ $P_2=(v_E \bowtie_{j_1} v_{C1}) \bowtie_{j_4} v_M$ $P_3=v_{ECM}$ $P_4=v_{EC} \bowtie_{j_4} v_M$
SELECT Ename, Job FROM Employee FROM Employee	$P_1=v_{ECJ}$	$P_1=v_{ECJ}$ $P_2=v_E \bowtie_{j_2} v_J$ $P_3=v_{ECJ} \bowtie_{j_2} v_J$ $P_4=v_{EC} \bowtie_{j_2} v_{ECJ}$

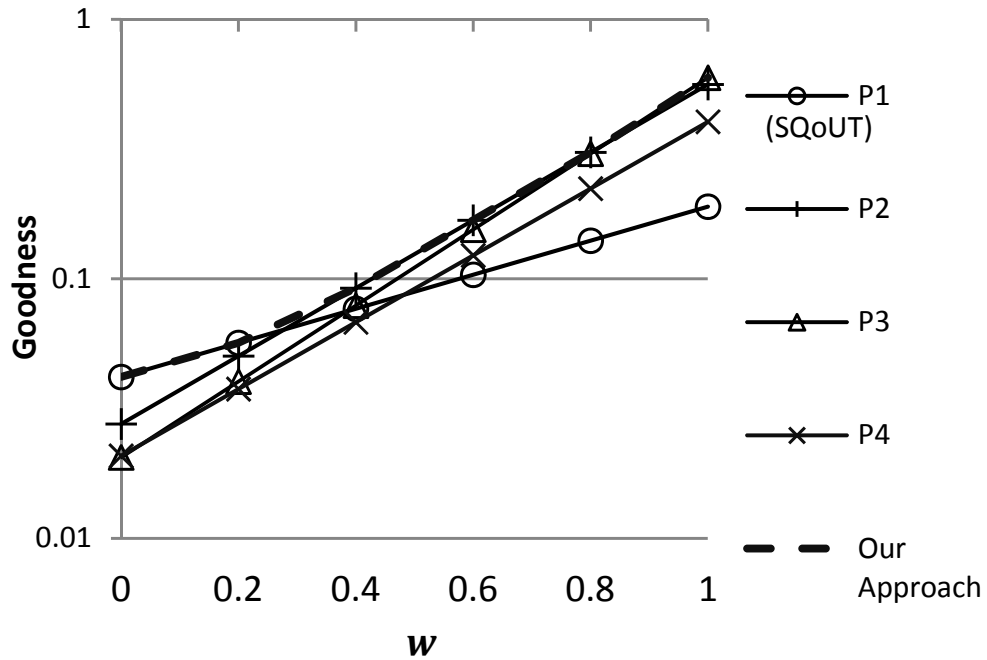


Figure 6.2: Average goodness of query plans (log scale)

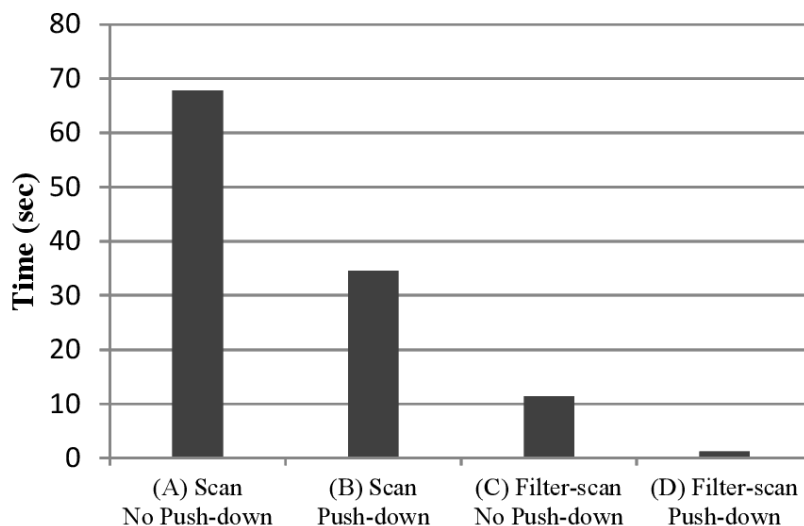


Figure 6.3: Query execution time in different settings

Table 6.5 shows two queries, and some of their possible execution plans under SQoUT and our system. SQoUT cannot find a plan to answer the first query since there are no available views to extract J_{obj} . For the second query, SQoUT has only one possible plan, while our system supports a wider plan space that may contain a better plan with higher goodness. Figure 6.2 depicts the average goodness of the four plans of query 2, with varying w . SQoUT selects Plan 1, and JITE is capable of selecting better plans since it considers a wider plan space.

6.3 System Analysis

In this section, we will analyze different components of the system. We study performance under different settings.

6.3.1 Optimization Benefit

We study the effect of the introduced optimization techniques, namely the document retrieval strategies, and join predicate push-down, that reduce the number of processed documents. Choos-

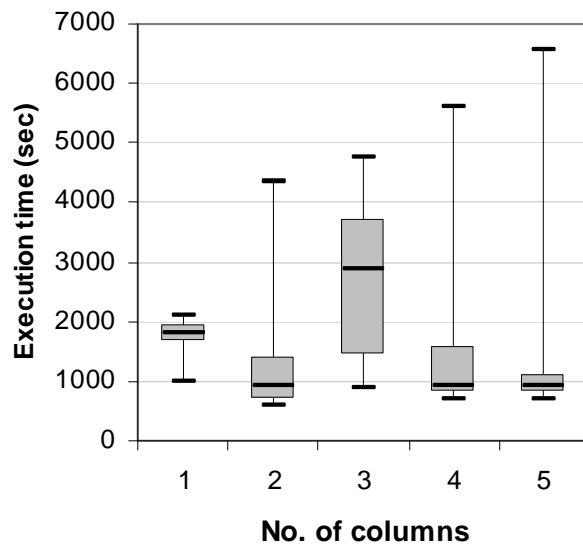


Figure 6.4: Referenced columns

ing from the two supported document retrieval strategies, and enabling or disabling the join predicate push-down gives a set of four settings. In this experiment we consider only the execution cost measured by the time taken to process the query. We assume the available views are $\{v_E, v_{C1}\}$. We run a manually designed query where all settings are applicable:

```
SELECT E.Ename, E.Ecomp
FROM Employee E
WHERE E.Ecomp='NBC' AND E.Ename = 'John'
```

The average query execution time under the four settings are shown in Figure 6.3. In setting (A), both views process all documents, and no optimization occurs. When enabling join predicate push-down in (B), the number of documents processed by v_{C1} decreases. In (C), *filter-scan* limits the documents processed by each view to those containing its corresponding keyword from the `WHERE` condition. This improves the performance, especially when enabling the join predicate push-down in (D), as v_E will check only documents in the output of v_{C1} .

6.3.2 Number of Extracted Attributes

Figure 6.4 shows a box plot (a graph depicting the smallest observation, lower quartile, median, upper quartile and largest observation) of the execution time, grouped by the number of columns referenced by each query. It can be seen that queries that reference fewer attributes execute faster, since only the IE systems that extract those attributes are invoked. For queries that reference the same number of columns, some finish faster than others due to the presence of selection predicates that filter out unneeded documents.

Chapter 7

Conclusion and Future Work

We summarize the work we have done and introduce possible extensions and future work that can extend our current approach.

7.1 Conclusion

The standard method of querying unstructured data is the Extract-Transform-Load (ETL) approach, but it suffers from decoupling in time and decoupling in space, and some applications cannot tolerate these drawbacks. In these applications, analyzing real-time data can be critical, and the schema might not be defined prior to extraction. In addition, the lineage information may be of interest to the application and the end users. Therefore, we introduced Just-in-time Extraction (JITE); an approach to define an extraction schema on-the-fly and perform extraction process online as part of query processing. We developed a system to allow users to utilize JITE to define interesting applications schemas during session time and to link them to the available information extraction systems. Users can pose SQL queries over the schema that reflects the structured information hidden in the textual data. The system extracts only query-related information from the underlying text corpus using state of the art IE systems. Since extraction happens during query processing, it is necessary to avoid extracting irrelevant information that is not going to contribute to the final query results. The query is analyzed, and the queried relations and query

predicates are employed to limit the extraction of relevant relations from relevant documents. The architecture of the system is inspired by the data integration Local-As-View approach. We visualize the user-defined schema as the global schema, and the available IE systems as views, called extraction views, over the global schema. We employ view matching algorithms to select a set of extraction views to answer the query at hand. The query optimizer matches and selects extraction views based on a defined cost model that considers the estimated extraction time and the quality of the extracted results. Different IE algorithms may take longer time to perform more analytic tasks and produce higher quality results. We allow users to specify their preference for extraction time or results quality. We developed a few query optimization techniques to minimize the number of documents processed in the extraction phase, based on the lineage of joining data and keywords from the SQL query. We implemented a GUI to communicate with the database, run SQL queries, show query answers, and visualize the origin of the extracted data.

7.2 Future Work

In our work, we focused on the basic functionality of extracting structured information from documents, and running SQL queries over the extracted data. There are many potential extensions that can add more functionalities and provide more optimization opportunities to JITE. This section suggests some possible ideas to extend this system.

7.2.1 Utilizing Properties of Specific Information Extraction Algorithms

In the current implementation, IE systems are treated as black boxes, ignoring the details of internal extraction algorithms. We used Apache UIMA to wrap IE systems, as it provides a unified view for extractors, and produces basic metadata for the extracted information. Users can plug a new UIMA extractor into the system, and the system considers it in the extraction process. Using a standardized framework for IE systems supports reusability of pre-developed IE systems that can apply any text analytics algorithms.

Having knowledge about how an information extraction system works, and about the details of its algorithm and its parameters, can help the system have more control over the extraction

process and can introduce more opportunities to enhance the performance, optimization, or quality of results. For instance, the system may be able to instruct the IE algorithm to restrict the output annotations to only those requested by the query, even if the IE system may be capable of producing more. Another possibility is to control a parameterized extraction algorithm to be more flexible (or more strict) when matching text, or comparing to a certain threshold. A simple parameter can be the document span to extract from. Limiting or widening the span window can have an effect on the labeling of entities, which can affect the extraction process.

7.2.2 Data Cleaning on Extracted Data

As mentioned before, IE algorithms are not perfect, and they usually produce some erroneous tuples. In the presence of integrity constraints over the extraction schema, an extracted relation may violate some constraints, and thus, require further cleaning operations to resolve the violations. The violations can be in the form of duplicate records, violations of functional dependency constraints, or other constraints. Data cleaning removes the violations that exist in a relation to make it satisfy a set of integrity constraints. There are many algorithms designed to clean a database instance and come up with some possible worlds or possible repairs [6, 5].

In JITE, data cleaning can be done as a post-extraction phase. The user can define some schema constraints during schema definition. The system can check if there exist any constraints defined on the queried relations. The extracted data is then checked for violations, and some cleaning algorithms can be used to return clean results. The work in [19] performs record linkage techniques as a post-extraction process, to get rid of noisy tuples. Cleaning extracted data can help improve the quality of query results; however, it is an orthogonal problem and can be tackled separately.

7.2.3 Probabilistic Information Extraction

In the relational model, data is defined to be certain. A tuple either exists, or does not exist. If it exists, it will be considered a trusted piece of information and will be considered in any relevant query. When performing extraction, however, some information cannot be certain to exist or

match a certain entity. For instance, Conditional Random Fields (CRF) build probabilistic models to segment and label sequence data [25]. The CRF can be trained to perform names entity recognition tasks to extract person names, locations, and organizations. The output of a CRF extractor is not certain, but probabilistic, labeling. For each token, the CRF framework assigns a set of possible tags with some probabilities. In general, the most likely label (the one with the highest probability) is considered to be the output of the extraction. However, the probabilistic tagging or extraction can have additional multiple benefits.

Probabilistic Query Answering

Querying uncertain data cannot be treated like regular relational data. Data can have probabilities or confidence on the attribute level or the tuple level (or both). When querying, we can output only tuples that are absolutely certain (with probability 100%), or those tuples that have a non-zero probability, or the set of tuples with a minimum probability threshold [29]. Usually, probabilistic queries run on probabilistic relations with no knowledge about how the probabilities were assigned to the tuples. In JITE, the system can utilize the extraction probabilities when answering a user-query.

Probabilistic Data Cleaning

Standard data cleaning techniques operate on certain tuples, but produce a set of probabilistic possible worlds (or possible repairs). New techniques of data cleaning can employ the fact that there may be more than one possible attribute value or joined record. For example, when the highest probability attributes values violate some integrity constraint, the system may choose to consider other values with lower probabilities.

7.2.4 Multi-Query Optimization: Caching

When processing a workload of queries, it is important to perform as much optimization as possible. Results computed for one query can be used later as part of an answer for a following query. This represents caching of pre-computed results for use in subsequent queries, providing

fast access to data, and avoiding re-extracting relations. Such an approach can be thought to be close to the concept of materialized views [16]. Query processing should consider the available cached views when finding the best plan to answer the query. In materialized views, the result of a view is cached to be re-used. In materialized views, when the base relations change, the cache stalls. Maintaining the cache of extraction views will face a few challenges. JITE originally targets applications that rely on fast changing data. Updating the cached results and incremental cache maintenance techniques can be relevant topics to consider.

References

- [1] Apache Derby, <http://db.apache.org/derby/>.
- [2] Yahoo! Pipes, <http://pipes.yahoo.com/>.
- [3] Eugene Agichtein and Luis Gravano. Querying text databases for efficient information extraction. In *ICDE*, pages 113–124, 2003.
- [4] Michele Banko, Michael Cafarella, Stephen Soderl, Matt Broadhead, and Oren Etzioni. Open information extraction from the web. In *IJCAI*, pages 2670–2676, 2007.
- [5] George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3:197–207, 2010.
- [6] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. *PVLDB*, 2:598–609, 2009.
- [7] Michael Cafarella, Christopher R, Dan Suciu, Oren Etzioni, and Michele Banko. Structured querying of web text: a technical challenge. In *CIDR*, 2007.
- [8] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. Join queries with external text sources: execution and optimization techniques. In *SIGMOD*, pages 410–422, 1995.
- [9] Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan Frederick R. Reiss, and Shivakumar Vaithyanathan. Systemt: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137, 2010.

- [10] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. GATE: An architecture for development of robust HLT applications. In *ACL*, pages 168–175, 2002.
- [11] Amr El-Helw, Mina H. Farid, and Ihab F. Ilyas. Just-in-Time Information Extraction using extraction views. In *SIGMOD (to appear)*, 2012.
- [12] Amr El-Helw, Ihab F. Ilyas, Wing Lau, Volker Markl, and Calisto Zuzarte. Collecting and maintaining Just-in-Time Statistics. In *ICDE*, pages 516–525, 2007.
- [13] Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: an experimental study. *ARTIFICIAL INTELLIGENCE*, 165:91–134, 2005.
- [14] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.
- [15] David Ferrucci and Adam Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10:327–348, 2004.
- [16] Ashish Gupta and Inderpal Singh Mumick. Materialized views. chapter Maintenance of materialized views: problems, techniques, and applications, pages 145–157. MIT Press, 1999.
- [17] Alon Y. Halevy. Answering queries using views: a survey. *VLDB Journal*, 10(4):270–294, 2001.
- [18] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. Here are my queries. Where are my results? In *CIDR*, pages 57–68, 2011.
- [19] Alpa Jain, AnHai Doan, and Luis Gravano. Optimizing SQL queries over text databases. In *ICDE*, pages 636–645, 2008.
- [20] Alpa Jain and Panagiotis G. Ipeirotis. A quality-aware optimizer for information extraction. *TODS*, 34(1), 2009.

- [21] Alpa Jain, Panagiotis G. Ipeirotis, AnHai Doan, and Luis Gravano. Join optimization of information extraction output: quality matters! In *ICDE*, pages 186–197, 2009.
- [22] Alpa Jain, Panagiotis G. Ipeirotis, and Luis Gravano. Building query optimizers for information extraction: the SQoUT project. *SIGMOD Record*, 37(4):28–34, 2008.
- [23] Alpa Jain and Divesh Srivastava. Exploring a few good tuples from text databases. In *ICDE*, pages 616–627, 2009.
- [24] Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. Systemt: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13.
- [25] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289, 2001.
- [26] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.
- [27] Dekang Lin and Patrick Pantel. Dirt - discovery of inference rules from text. In *SIGKDD*, pages 323–328, 2001.
- [28] Andrew McCallum. Information Extraction: distilling structured data from unstructured text. *Queue*, 3:48–57, 2005.
- [29] Jian Pei, Ming Hua, Yufei Tao, and Xuemin Lin. Query answering techniques on uncertain and probabilistic data: tutorial summary. In *SIGMOD*, pages 1357–1364, 2008.
- [30] Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942, 2008.
- [31] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

- [32] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. Damia: data mashups for intranet applications. In *SIGMOD*, pages 1171–1182, 2008.
- [33] Mohamed A. Soliman, Mina Saleeb, and Ihab F. Ilyas. MashRank: towards uncertainty-aware and rank-aware mashups. In *ICDE*, pages 1137–1140, 2010.
- [34] Matthew Solomon, Cong Yu, and Luis Gravano. Popularity-guided top-k extraction of entity attributes. In *WebDB*, 2010.
- [35] Clement Tak Yu and Cheng C. Chang. On the design of a query processing strategy in a distributed database environment. In *SIGMOD*, pages 30–39, 1983.
- [36] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.
- [37] Patrick Ziegler and Klaus R. Dittrich. Three decades of data integration – All problems solved? In *WCC*, pages 3–12, 2004.