INTEGRATING AND ANALYZING DATABASES AND INTERRELATED DOCUMENTS

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Carlos Garcia Alvarado

December, 2012

INTEGRATING AND ANALYZING DATABASES AND INTERRELATED DOCUMENTS

Carlos Garcia Alvarado

To my wife, Johnna, and our family and friends.

INTEGRATING AND ANALYZING DATABASES AND INTERRELATED DOCUMENTS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Carlos Garcia Alvarado

December, 2012

Abstract

The velocity, variety, and volume of present-day data bring about new problems that Database Management Systems (DBMS) must handle. In particular, text data encapsulate the essence of the so-called “unstructured data” and the need for efficient in-database algorithms and data structures for their analysis. Multiple solutions have been proposed for preprocessing, integrating, and analyzing heterogeneous sources via ad-hoc systems. This dissertation defends the idea that text corpora can be managed efficiently inside a relational database management system via SQL and database extensibility mechanisms. It presents data layouts and algorithms for preprocessing, storing, and querying text data efficiently within a DBMS. The optimizations focus on one-pass algorithms, pushing in-memory computations and reducing the number of I/Os. Furthermore, the DBDOC project introduces a new algorithm and properties for integrating and querying structured information stored in a relational database management system and the unstructured data living outside the DBMS realm. In a complementary manner, an original query recommendation algorithm based on OLAP cubes enhances the querying system for finding related concepts. The last chapter of this dissertation is based on exploring heterogeneous data analysis in the form of text corpora stored within a DBMS. The results of this research are a couple of original OLAP-based algorithms for extracting knowledge from text corpora. ONTOCUBE and CUBO focus on extracting and generating OLAP cubes from ontologies, respectively. The distinguishing trait of both algorithms is that they exploit the sparse nature of text data and perform efficient frequency summarizations. In addition, ONTOCUBE presents new measurements for building ontologies using OLAP cubes and CUBO formalizes the notation to map the hierarchy behind an ontology to compute multidimensional aggregations on classified documents. Finally, all of these document data exploration algorithms are then refined and adapted for exploring source code files that reference a database schema. This dissertation concludes with important open problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The data generation, acquisition, and affordable storage for massive data have pushed traditional relational database systems to the limit [21]. This is especially evident when working with large repositories of heterogeneous data generated quickly, such as text, biological data, or stream data, since these require the generation of new algorithms for efficient processing and knowledge discovery [81].

This creation of new algorithms for managing the new volume, velocity, and variety of data has been approached in areas such as Database Management Systems, Information Retrieval, Data Mining, and High Performance Computing. Information retrieval has focused on effective algorithms for finding relevant information from text corpora [55, 22], Data Mining has focused on the discovery of new information about the data [59, 115], and Database Management Systems and High Performance Computing have focused on efficient algorithms for managing, processing, and retrieving large data repositories [48, 28].

The definition of “large” and “big” are subject of debate among these scientific communities. For the purpose of this dissertation, “large” is considered to be relative to the computational system. Hence, a repository is considered to be “large” when the type and amount of data exceed the capabilities of a system, creating a need for “smart” algorithms that are able to process these data efficiently within the constraints of the system. For example, a text collection of only a few hundred documents in a personal computer is considered a“large data repository” with respect to how it is considered in a large cluster of hundreds of computers.

These large heterogeneous sources are difficult to query and explore not only because of their volume, but also because of their characteristics. These properties are exploited in the high-speed generation of stream data or the high dimensionality of text corpora [7, 49, 40]. Some of these characteristics are also shown in data that have some structure or were generated from an structured source difficult to manage. Additional motivating examples of these types of heterogeneous sources include medical data, digital libraries, emails, social networking data, and phone data streams, among many others.

Structured data has traditionally been stored in relational database management systems (DBMS) due to these systems’ unique capabilities for guaranteeing data integrity and persistence [28]. Also, DBMSs are tuned for efficient data management of large data sets and provide flexible querying. Unfortunately, traditional DBMSs cannot guarantee the same efficient performance when working with semistructured and unstructured data. This limitation has led to the development of a myriad of ad-hoc implementations for managing each type of data [26], in which each implementation includes only a small subset of the functionalities that are already part of a database management system.

Numerous companies and researchers have identified the need for a unified platform that is able to manage structured and unstructured data under the same umbrella [113, 81]. This 'ideal' scenario provides multiple advantages during the information retrieval/data mining processes, because it empowers an organization to gain more control over its data and offers the possibility of generating more revenue from it. Therefore, this dissertation defends the idea that it is possible to manage, in an efficient manner, semistructured and unstructured data from inside a database management system by extending the functionalities of a relational database management system.

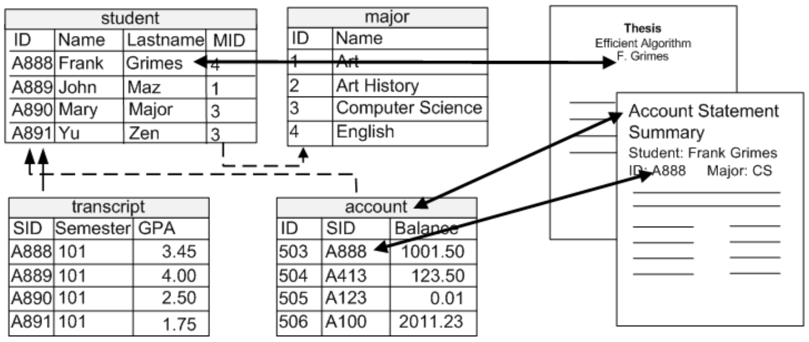
Multiple ad-hoc solutions have been proposed that attempt to manage the data resulting from data preprocessing, integration, and knowledge discovery problems. However, as has been noted by [58], integration and management of heterogenous data is still in its "teenage years", and these systems lack the flexibility and robustness provided by a DBMS. Therefore, modern database systems require hybrid capabilities to satisfy the necessities of contemporary users regardless of the development of ad-hoc solutions for all their data needs [113]. Extending a database system should allow maintaining the separation that exists between the logical and physical data model in order to support flexible querying.

Different systems and query languages have been proposed to manage heterogenous data preprocessing, exploration, and knowledge discovery within the information retrieval (IR) and the database systems (DB) community. However, both communities are still far from obtaining a unified solution.

1.1 Advantages of a DBMS-based Approach

Database management systems (DBMSs) have been well recognized for being key players in large scale data management and processing [97]. Advantages of such systems include efficient data entry, amortized management and retrieval costs, data redundancy, data integrity, and data security. In addition to these advantages, DBMSs allow a clear separation between the logical and physical data layers. This separation is crucial for several organizations' goals, as well as for our research objectives, because it is possible to store and manage the data without the need to compromise our knowledge discovery and exploratory power. As such, newly developed applications enjoy efficient data storage and retrieval without the need to explicitly enforce data integrity, security and persistence.

Given all of these advantages, this dissertation defends the idea that the DBMS can be extended to effectively and efficiently manage structured and semistructured sources in an integrated fashion. Thus this research focuses on the problem of extending the database management system's capabilities for searching and exploring structured and unstructured sources based on new algorithms that rely on SQL queries



student

ID	Name	Lastname	MID
A888	Frank	Grimes	4
A889	John	Maz	1
A890	Mary	Major	3
A891	Yu	Zen	3

major

ID	Name
1	Art
2	Art History
3	Computer Science
4	English

transcript

SID	Semester	GPA
A888	101	3.45
A889	101	4.00
A890	101	2.50
A891	101	1.75

account

ID	SID	Balance
503	A888	1001.50
504	A413	123.50
505	A123	0.01
506	A100	2011.23

Figure 1.1: Example of a Registrar's Office Database and Corpus.

and database extensibility functions (e.g. user-defined functions or stored procedures). In particular, we focus on the subproblems of preprocessing unstructured data inside a DBMS seen as a collection of documents stored, and data integration and knowledge discovery. Moreover, we present new algorithms to efficiently solve data preprocessing inside a DBMS, and to perform integrated searches, exploration, and knowledge discovery on heterogeneous data.

1.2 Motivation

There are multiple domains which require a unique solution for managing heterogeneous data. These areas include, but are not limited to, medical data, digital libraries, social media, geographic information systems, and networking communication, among many others. Currently, as stated in [16], almost any application that manages documents and text requires the joint capabilities of IR and database systems in order to provide the users with a complete application [113, 103]. This integration allows the users to efficiently answer questions which in a non-integrated scenario would be impossible to answer.

A typical example of heterogeneous data that requires integration can be seen in a collection of documents (a semistructured source) created from a structured source such as the registrar's office at a university. This example is detailed in Figure 1.1, in which a central relational database containing information about current students in the university, is maintained, including student ID, name, major, grades, and financial information. In parallel to this database, each department and professor has documents about these students, such as medical information, assignments, reports, theses, and papers, among many others. These unstructured sources are all modified and created independently, without the database support. Nevertheless, there is a clear connection between all of these new documents, but the references between both sources are no longer obvious, and there is an increasing need to perform searches that will allow us to relate and rank these data sources again. As can be seen in this example, the keywords "account", "student," or "transcript" represent three major classification topics of the documents contained in the text data set. As a result, a document that contains any of these keywords is likely to have some relation to any of these topics. As we analyze additional elements in the database, for example, student ID, last name, first name or email, some additional references can be inferred between the DBMS and documents in the corpus that lead to more complex searches. Some of these complex questions include:

- "What is the average number of publications for graduate students with a GPA above 3.5?",

- "What are the most popular publication topics for students with a GPA above 3.5?",
- "What are the most popular research topics for students that are studying Art History?"

Preprocessed and integrated sources allow us to perform complex searches that can answer non-trivial questions and promote the discovery of information. For example recommending related concepts for Computer Science, such as "Networking" or "Game Programming" when a user query is performed. However, these types of searches require complex numerical computations that traditionally have been performed outside the DBMS due to the inability to have arrays in a DBMS. In previous research [89], it has been proven that it is possible to perform these computations inside the DBMS without needing to export the data outside of it, thus avoiding the I/O bottleneck. Therefore, it is necessary to present scalable algorithms that can perform knowledge discovery tasks for this type of data within the relational database management system.

1.3 Challenges

Despite the fact that the advantages are clear, a number of challenges arise for discovering knowledge in preprocessed and integrated data sets [5, 104, 108]. In order to tackle these problems. This dissertation presents the topics as follows:

- data preprocessing

- data integration
- data exploration
- knowledge discovery

Data preprocessing has traditionally been performed outside relational database management systems due to the need for ad-hoc data structures that allow building inverted indexes and stemming keywords, and which provide efficient access to the preprocessed data. However, new Extract-Load-Transform approaches in the database community (see [111]) have inspired our work to perform data preprocessing tasks after the data has been loaded to the database management system. Some of these transformations include removing irrelevant keywords, keyword stemming, inverse index construction, and statistics and data precomputations using SQL, user-defined functions and various data layouts.

Data integration algorithms focus on finding relevant information across heterogeneous sources. In order to do so, the algorithm must provide a way to identify the correspondence between elements among all the domains. In addition to this, the efficiency of these algorithms is of the utmost importance due to the size of text corpora.

An advantage of a preprocessed and integrated set of data sets is the ability to query and explore them. As a result, efficient and scalable algorithms have to be provided that allow retrieving non-trivial information for the users. Some examples of these algorithms include boolean searches, traditional searching models (such as the

Vector Space Model and Probabilistic Models), top-k searches, and Online Analytical Processing (OLAP) techniques for data set summarizations.

The final goal of this dissertation is to provide algorithms that allow for knowledge discovery in integrated sources. Some of these knowledge discovery techniques include OLAP algorithms that have to deal also with the high dimensionality of text corpora.

Therefore, the document defends the idea that it is possible to explore, search, and in some cases, rank the integrated data that exist among heterogeneous sources residing in a relational database management system. As a result, some typical Information Retrieval (IR) and Data Mining capabilities will be provided to the user for querying and exploring medium-sized collections of interrelated documents with an acceptable performance under the same umbrella as a relational database system. Hence, the objective of allowing flexible querying while maintaining the physical and logical data independence that is key for a DBMS.

1.4 Chapter Roadmap

The organization of this dissertation is as follows: Chapter 2 introduces the common notation and basic definitions to be used for referring to the structured and semistructured sources. Due to the interdisciplinary nature of this work, definitions are also provided from the information retrieval community community to address information presented regarding data retrieval, exploration and knowledge discovery. Background concepts required to better comprehend work presented in this dissertation are also presented.

In Chapter 3, research related to this dissertation is presented. In addition, our contributions in every area are briefly described (a more thorough description of contributions is contained within the introduction and conclusions of each chapter.) Chapter 4 explains proposed algorithms for preprocessing efficiently text data inside a relational database management system, which include experimental evidence to support our claims. This dissertation also addresses future work and conclusions, and is based on the work presented in [39, 45]. Chapter 5 presents our work in data integration, which includes finding references between the already preprocessed unstructured data and the relational data already present within the DBMS. In addition, these integrated data sources are queried using traditional IR techniques adapted to work efficiently inside a DBMS. Finally, OLAP techniques are adapted to work efficiently with these text corpora and recommend related searches. This chapter condenses research in [39, 45, 40, 46, 41]. Chapter 6 contains research regarding Knowledge Discovery within these preprocessed data sources. Some of this research includes ontology generation, text data exploration using data cubes, and source code exploration as a particular case from the document exploration approaches presented in this disseration. This chapter is the result of the most important findings in [101, 20, 36, 94, 18, 40, 97, 102, 38, 41, 92, 96, 42, 43, 44, 47, 95]. Chapter 7 presents a final model for preprocessing, integrating, and querying unstructured data. This model encompasses a conceptualization of all of the work presented in Chapters 5 and Chapter 6. Finally, Chapter 8 presents the overall conclusions of this dissertation and analysis of future directions.

Chapter 2

Background

The interdisciplinary nature of this dissertation requires unifying concepts from information retrieval and data mining. In addition, all concepts are unified under a common notation. First, we introduce concepts for managing structured data and unstructured data. Next, the definitions for ranking, exploring, and discovering knowledge on the preprocessed and integrated sources are presented.

2.1 Definitions

Let a central database D be stored in a DBMS and a collection of unstructured sources of text data (corpus) be C. As shown in Figure 1.1, the central database represents the structured data stored under a schema in a relational database system, and the set of theses, account statements, medical records, and migratory information represent the unstructured data that holds some relation to the structured information. It is important to notice that even though there is an underlaying structure behind all the unstructured documents, this information is unknown and the structure is not shared across all of the documents in the collection.

2.1.1 Database

Let this database D have $\{T_1, T_2, \ldots, T_m\}$ tables, where m is the number of tables in D. In addition, each T has $|T_j|$ tuples or rows. Notice that j represents a subscript from a table in D. Furthermore, each T_j contains attributes defined as $T_j(a_{j1}, a_{j2}, ...)$. In addition, there exist relationships between tables of the form $T_i(a_{i1}, \ldots) \rightarrow T_j(a_{j1}, \ldots)$ given by primary/foreign key referential integrity constraints.

2.1.2 Documents

Let a collection of documents (semistructured data) in a corpus be defined as $C = \{d_1, d_2, \ldots, d_n\}$, where n is the number of documents in the collection. Similarly to the structured domain, i represents a subscript that identifies a particular document in C. A document d_i is composed of a set of keywords k, where every k represents a string that can be mapped to one or more words or concepts. In this dissertation, it will be common to have a table T_j or an attribute a_{j1} be mapped ($\mapsto$) to a keyword k, which should not be confused to the PK/FK notation.

A result of the algorithms to be presented here is that the user is allowed to query the heterogeneous sources. A query q or Q is defined as a set of keywords

$q = \{k_1, k_2, \ldots\}$, where every k is a keyword or concept as defined previously. The result of a query can be ranked based on different scoring functions (or retrieval models).

2.1.3 Information Retrieval

Retrieval models assign a similarity coefficient between a given query q and a document d_i. The retrieval models (for further reference consult [55]) are summarized as follows:

- Boolean Model: Documents that contain all (for conjunctive queries) or some (for disjunctive queries) of the keywords in q.
- Vector Space Model: The documents and the queries are managed as vectors of k.
- Probabilistic Retrieval: These models are based on the probability of a k appearing in a relevant document.
- Language Model: The similarity coefficient is the result of the likelihood that a document will generate the given query.
- Latent Semantic Indexing: Singular Value Decomposition (SVD) is used to reduce the dimensionality of the text corpora and to associate documents with similar semantics.
- Interference Networks: A Bayesian network is used to obtain the relevance of a document given a user query.

- Neural Networks: A set of neurons is used to weight the document given a query.
- Genetic Algorithms: Document scoring is generated by the evolution of estimated weights.
- Fuzzy Set Retrieval: The documents are mapped to fuzzy sets, and a strength of membership is associated with each document.

We decided to work with the Vector Space Model, and two alternative methods that encapsulate the most fundamental trends for information retrieval (the other one being Page Rank): Okapi Probabilistic Model and the Dirichlet Prior Language Model. In addition, these models are used because they match directly on keywords. Some other ranking models, such as Latent Semantic Indexing (LSI), assume that direct keyword matching is a point of failure and reduce the number of dimensionalities using singular value decomposition (SVD). Notice that the LSI method is an extension of the Vector Space Model with the entry data set transformed by SVD. As a result, the Vector Space Model, Okapi, and Dirichlet Prior, represent the Vector Space, Probabilistic Retrieval, and Language Model strategies, respectively. Therefore, these three scoring models share a common framework that can be captured in a database management system, and the types of aggregations and optimizations can be shared among these ranking functions. Let Sim represent a function that takes a query q and a document d_i and returns a similarity score for such pair. In addition let function $tf(\cdot)$ return the frequency of the given keywords in a document. Furthermore, let a function $df(\cdot)$ return the frequency of a keyword in a document

d_i. Then, let function $idf(\cdot)$ to represent the inverse document frequency given by 2.1.

$$idf(k) = \ln \frac{n - df(k) + 0.5}{df(k) + 0.5} \tag{2.1}$$

The Vector Space Model [106, 56], which takes all documents related to a given query and then obtains the rank of all of these related documents is the most basic of this methods. Moreover, it is possible to extend this method to other applications. This ranking is generated by taking the frequency vector of the given query and every document containing any searched keyword. These frequency vectors are ranked by applying a Similarity Formula (see Equation 2.2). In this formula, the keyword vectors of the given query (q) and the i^{th} *document* (d_i) are computed with a dot product between them and then divided by the norm of the q vector and the d_i vector.

$$Sim(q, d_i) = \frac{\overrightarrow{q} \cdot \overrightarrow{d_i}}{\|\overrightarrow{q}\| \left\|\overrightarrow{d_i}\right\|} \tag{2.2}$$

The Okapi Probabilistic Model scoring function has been recognized as an effective retrieval method for abstracts and titles (see Equation 2.3), although it has to be adjusted by setting some constants: c_1, c_2 and c_3 as presented in [32]. The OPM requires additional computations such as the average length of a document (avglen). This formula may produce negative values when the frequency of the keyword in the collection is greater than half of the number of total documents in the collection.

$$\begin{aligned} Sim(q,d_i) = \sum_{k \in q \cap d_i} idf(k) \quad &\times \quad \frac{(c_1+1) \times tf(k,d_i)}{c_1(1-c_2) + c_2\frac{|d|}{avglen}) + tf(k,d_i)} \\ &\times \quad \frac{c_3 + 1 \times tf(k,q)}{c_3 + tf(k,q)} \end{aligned} \tag{2.3}$$

The Dirichlet Prior Language Model (DPLM) is a language model scoring method that is based on the probability that a document will "form" a given query (see Equation 2.4). Therefore, it takes into consideration the probability of occurrence of a keyword in the collection, although, DPLM has a μ parameter for adjusting the method based on the characteristics collection [32, 55].

$$P(q|d_i) = \prod_{k \in q \cap d_i} \frac{tf(k,d_i) + \mu P(k|C)}{\sum_k tf(k,d_i) + \mu} \tag{2.4}$$

2.1.4 Top-k Algorithms

The rank operation is holistic; as a result, the whole data set is needed to obtain correct results. Despite this, top-k algorithms are used to avoid visiting and evaluating all of the elements in a set of lists that are combined by a given function. The interest in Top-k algorithms was brought on by different algorithms that have been explored in numerous recent papers [66, 31, 25, 67, 30, 68, 31]. The most representative ones are the TA algorithm [30], the NRA algorithm [31], the Rank-Join algorithm [68], and the J* algorithm [87].

We focus on ranking documents existing within a DBMS, adding new elements to take into consideration compared to the ones described in the ranking models. We are interested in describing the access of the tuples in the database. The Rank-Join algorithm was chosen because it is possible to pipeline it, extend it to the relational data layout, and use it with a monotonic aggregation function with conjunctive queries. We decided that the Rank-Join algorithm is more suitable for being implemented in SQL and as a UDF because one only needs to keep track of the first element on the ordered list and the last seen element. Therefore, we avoid having to reach other elements on the list at a time.

Let L_h be a ranked element h in the list L, which could represent a result table from a previous Rank-Join step, or a new set of m input tables. Additionally, let q represent a given query by the user, and p, the number of different keywords that composes that query. $F(t_1, ..., t_p)$ represents the ranking function (in this case, the Vector Space Model formula) based on the weights of the keywords and the frequency of the keywords in the document. T is a safe cutoff value to report values in the top-k elements and $t^{(max\ or\ last-seen)}$ represents the maximum score of a keyword, on the list, or the last seen score of t in L, and J represents joint tuple given that join condition is true.

2.2 Data Integration

Let A_k be a set of all the approximate keywords of k given that the edit distance of keyword k' from keyword k is $\leq MAX(1, \beta * |k|)$, where β is a real number between 0 and 1. Intuitively, the value for β cannot be larger than 0.5 because it will accept many unrelated keywords as valid approximations. Still, this value can be tuned as required. In the particular case in which exact matches are the only ones desired, the edit distance between k and k' should be zero instead.

In addition, let e be an element stored in D in a level of granularity γ. Moreover, let the granularity level γ be represented by a value representing a table, a column in a table, or a record in T_j. As a result, an element in the database e with a granularity level representing a row will require a pointer in the database to be a set of primary keys and column references, or a reference to a column in the column level, or a reference to a table in the table level. For example, an element in the row granularity level is represented as $e =< A888, \{T = account, c = SID, PK = 503\} >$. Even though entity resolution is not the focus of this dissertation, different machine learning (ML) algorithms or natural language processing (NLP) techniques can be applied to map an element e in D to a keyword [1]. Without loss of generality, we assume that the elements of the database are the keywords extracted from the table and column names (metadata) as well as the rows in a table (actual data content).

Moreover, let δ be the edit distance and let *EditDistance(k,k')* be a Boolean function that returns true if k' is an approximate keyword match of k.

2.3 Cube Exploration and Knowledge Discovery

Exploring and ranking integrated heterogeneous sources provides relevant information to the user. However, it is possible to extract new information for the new user that is not trivial. In order to do so, we exploit the OLAP algorithm for extracting new knowledge from a preprocessed source. Finally, an application that exploits all of the techniques presented here is used in the context of source code analysis.

2.3.1 OLAP Cubes

OLAP techniques are generally used to quickly process complex queries involving multiple aggregations [54]. While traditional applications of OLAP include summarizations in the form of business reports and financial analysis of large data sets [19], we believe that the OLAP dimensional lattice can be used to efficiently summarize text corpora. The main benefit of OLAP is the ability to analyze large amounts of data at various levels of aggregation in order to obtain additional information. In this case, the dimensions are represented by the collection of classes, and the attributes are measurements relating a document and a class, while the level of aggregations (ontology) allows one to obtain various combinations of the parents of the leaf dimensions.

Traditional OLAP accepts inputs that are horizontal, which means that each row contains all dimensions used in the analysis.

However, this dissertation proposed that the fact table has a vertical layout, which translates into each row containing a single dimension. This presents a challenge for the algorithm presented here because normal techniques, such as slicing, cannot be used with the vertical layout. Despite this, only a small subset (the subset of dimensions in Q) of all dimensions in the corpus is analyzed. It is nevertheless time consuming to pivot the dimension into a horizontal layout. This is especially costly when the number of documents is large and the dimensions are sparse. As a result, an algorithm was developed that can obtain the necessary aggregations and perform the required auxiliary computations by taking only the desired dimensions. Likewise, the entire data cube for all existing combinations is computed with no minimum threshold required.

OLAP data cubes are able to capture hierarchies in the data. These hierarchies can be described as flat, balanced, or unbalanced. The most basic type of hierarchy is flat where only the leaf nodes are present. A second type of hierarchy is balanced, where all the nodes are present for every level of the tree. Finally, unbalanced hierarchies are those that do not have nodes in every level of the hierarchy. All of these types of hierarchies can be represented in an OLAP data cube. However, unbalanced hierarchies require additional information in the ontology, such as providing a hierarchy level for each class as a property. Despite these advantages for managing hierarchies, it is not possible to model every hierarchy variant, such as a node with multiple parents. Multiple parent hierarchies cannot be represented in a transparent manner. In this case, a separate hierarchy is needed to model every parent-child relation.

2.3.2 Cube Exploration

As before, let C be a collection, or corpus, of n documents $\{d_1, d_2, \ldots, d_n\}$. Each document d_i or query q is composed of a set of keywords k. In addition, let x_i be a frequency vector of all the different keywords in C for each document d_i. The collection is stored in an inverted list format within the table vtf that contains the document id, keyword and position in the i^{th} document. A summarization table tf is computed from vtf as a table with the document id i, the keyword k, and the keyword frequency f stored. In addition, let $\hat{d}$ be a subset of documents, $\widehat{tf}$ a subset of keywords from tf, and $\widehat{vtf}$ a subset of vtf.

The mathematical representation of the OLAP data cube is the dimensional lattice, which has a size of $2^{\hat{t}}$, where $\hat{t}$ is the number of keywords. One level (or depth in the lattice) of the cube is denoted by $\binom{|\hat{t}|}{level}$, such that $level \leq \left|\hat{t}\right|$. A similar algorithm for computing combinations is the A-priori algorithm. For the purpose of this dissertation, let the transactions T be equivalent to documents, the itemsets be equivalent to keywords, and the support measure considered as the frequency of keywords in a document.

2.4 Ontologies

Ontologies embody formal representations of knowledge. As such, several language proposals have been used to write explicit formal conceptualizations of domain models [3]. These proposals include the popular RDF/RDFS, as well as some other

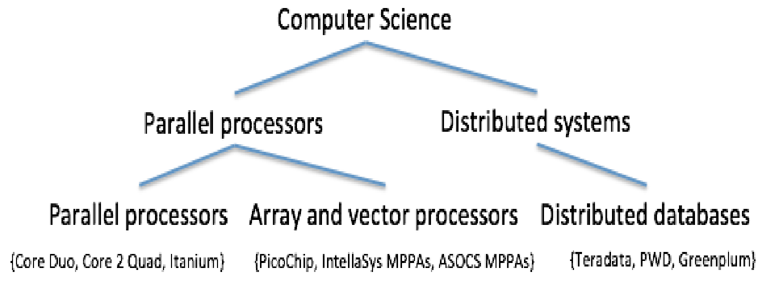

Computer Science
Parallel processors
Distributed systems
Parallel processors
Array and vector processors
Distributed databases
{Core Duo, Core 2 Quad, Itanium}
{PicoChip, IntellaSys MPPAs, ASOCS MPPAs}
{Teradata, PWD, Greenplum}

Figure 2.1: Ontology Example.

representations such as DAML +OIL and OWL languages, among many others. RDF/RDFS allows some basic knowledge representation. Despite this, features such as the local scope of properties, the disjointness of classes, Boolean combinations of classes, cardinality restrictions, and special characteristics or properties are left out of this language [3]. In order to model more complex relations between classes, DAML+OIL took an object-oriented approach [62]. Finally, OWL was developed to standardize all the previous language approaches in a more robust knowledge representation.

Regardless of the ontology language, all of these knowledge representations rely on the specification of the main concepts in a given context. The representation of every concept is defined as a class with a set of properties and interactions between the classes, subclasses, and their properties. A class (e.g. owl:Class) is a classification of individuals that share common characteristics. This class classification is obtained through a taxonomy (hierarchy). In this work, CUBO is not tied to a specific language, but on the classes (dimensions) and their relationships. An example of a class taxonomy is given in Figure 2.1, in which the ontology used for the data cube example is shown. The first level corresponds to concepts D_1 =Parallel processors,

D_2 =Array and vector processors, D_3=Distributed databases; the second level to concepts $H_{(1,1)}$ =Parallel processors and $H_{(1,2)}$ =Distributed systems; and the root level (H_0) is for the Computer Science concept. Each node in the taxonomy is a class, and the branches in the tree indicate a type of relationship (e.g. subClassOf). The instances represented in the leaf classes are specific to a particular class (which may or may not be considered as part of the data cube).

Due to the hierarchical nature of ontologies, there exists a mapping with OLAP. Let a collection, or corpus, of n documents, where each document is redefined as t_i and composed of a set of dimensions $\{D_1, D_2, \ldots D_j, \ldots, D_k\}$, where k is the number of the dimensions required for building the data cube. Examples of dimensions in a corpus are the topics "Parallel Processor" or "Distributed Database". Moreover, let every set of dimensions in a document be accompanied by a set of attributes $\{A_1, A_2, \ldots, A_m, \ldots, A_e\}$, where e represents the total number of attributes and A_m is a specific attribute (e.g. minimum common frequency of a set of attributes). Moreover, let the collection be stored in a relational table inside a DBMS. Thus, F is a table defined as a vertical fact table $F(i, D_j, A_1, \ldots, A_e)$ of size $e + 2$ by $\sum_{i=1}^{n} |t_i|$, where i represents the document number, D_j is a dimension, and A_m is a summarization attribute. A_1 to A_e are constant within every D_j of a document to avoid generating a much larger fact table. Notice that F differs from a traditional fact table for OLAP, because the dimensions are not represented in a horizontal manner and all of the attributes are replicated within every D_j. This is due to the sparse nature of text data and the limitation of the number of columns that is present in a traditional row store DBMS. Additionally, if the attributes are not considered to be constant within the same document, additional processing within a document must be performed to obtain a unique aggregation measurement. Since the OLAP dimensional lattice has a size of 2^k combinations for a set of dimensions, where k is the number of dimensions. Furthermore, a user-query Q is a subset of dimensions from F which builds an OLAP data cube for every level in a given dimension hierarchy represented by the l subscript.

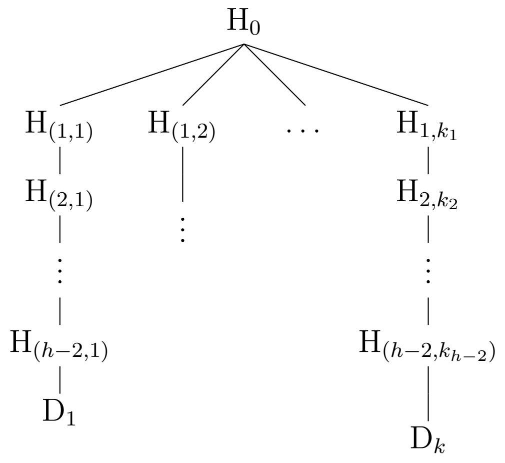


Figure 2.2: Ontology.

An ontology $\mathcal{O}$, on the other hand, is mapped to a dimension hierarchy as a tree-like structure where the nodes represent dimensions and the branches model the relationships between them. In Figure 2.2, the k leaf nodes D_j represent the ground level dimensions and H_{h-2} to H_0 represent the parent dimensions, where h is the depth or height of the ontology. In addition, the number of dimensions per level k_l is always less than or equal to the following $k_{(l+1)}$. Lastly, we will consider an ontology to have an overall H_0 root class (e.g "thing"), and every dimension to have only a single parent (except for the root that has none). In relational terms, every concept reference between levels (e.g. D_j and H_{h-2}) can be seen as a dimension table. However, in this approach, the entire data structure is kept in main memory (to be detailed in sections to follow) as a tree during the execution of the algorithm.

2.5 Querying and Programming

A database management system offers different mechanisms to access the data. Structured Query Language (SQL) is a special-purpose language designed to provide access to the data without the need to know the storage details of the underlying data (the physical design). SQL takes advantage of the schema of a database to manage the data. SQL allows flexible querying without the need to provide a new data accessing infrastructure.

Other mechanisms, such as user-defined functions (UDFs) and stored procedures (SP) offer the possibility to incorporate extensions to a database management system. UDFs and SP provide the advantages of structured languages.

2.5.1 SQL Queries

SQL is a declarative language designed to query the data in a DBMS. SQL is based on relational algebra and provides ways to modify the schema of the data, as well as insert, update, and delete information and security privileges. The SELECT statement is the most important because it retrieves the data from one or more tables and allows the execution of computations or the calling of extensibility mechanisms. SQL statements provide native parallelism and are portable, in their majority, across

DBMS.

There are, unfortunately, important limitations in the SQL language. SQL has a limited ability to manage scientific computations, and heterogeneous data processing. These computations are normally related to the lack of data locality exploitation, the lack of vectors and matrices, as well as the possibility of exploiting ad-hoc data structures.

2.5.2 SQL Extensibility Mechanisms (UDFs)

Extensibility mechanisms such as user-defined functions (UDFs) and Stored Procedures (SP) are constructs that allow exploiting main memory in an ad-hoc manner using object-oriented or Procedural Languages constructs. user-defined functions are divided as scalar, aggregate, and table-valued functions. On the other hand, Stored Procedures can be considered equivalent to table-valued functions for the purpose of the research presented in this dissertation, even though they are not equivalent, since an SP is not an expression.

Aggregate user-defined functions (aggregate UDFs) give users the capability to extend the functionality of the DBMS [13]. The set of steps (see Figure 2) that must be implemented by the user to program the aggregation are:

- Initialize: Data structures and variables for the aggregation are initialized.
- Accumulate (or Aggregate): This step is the most important. In this step, each row of the data set is processed, one at a time. An accumulation in a

local variable is performed by every thread. Notice that while the table scan is being processed, the threads are fed the rows to accumulate.

- Merge (or Combine): This step merges the accumulated values of independent threads into a single thread. This thread is responsible for merging both local variables and local data structures into a global aggregation result.

- Terminate: In this final step, after all of the threads partial results have been merged, the function return value is computed. Once this last step is finalized, the global aggregation result is returned to the user.

It is important to point out that due to the fact that user-defined functions are compiled fragments of C code, the input arguments for aggregate functions must be fixed in order to allocate memory space and allow argument value passing to each thread [89]. Therefore, to allow a dynamic dimensional vector as an argument, the values for the attributes of a data point have to be packed: either a string or a binary object. Execution performance of the aggregate UDF can be improved by materializing a table with a single column, storing the packed dimensions with the user-defined type. Even though the aggregation can be computed efficiently when data are already in binary format, creating such a table is a pre-step which can be time-consuming, especially for large input tables.

Table-valued functions (TVFs) are a type of user-defined function that, unlike aggregate UDFs, can return a table as the final result of the function. TVFs read an input data set as a single data stream and do not implicitly manage parallelism. Despite the lack of “out-of-the-box” parallelism, it is common that database systems allow the user to implement routines that support parallelism. In this dissertation, a TVF will be used to compute multidimensional aggregations, with internal thread management algorithms returning a result table with just one row.

2.6 Integration and Querying of Programs and a Database

As an extension and area of application of the work presented here, source code analysis appears as a natural extension. A database D contains all of the elements that we would like to match against the source code of a program to find non-trivial information. A program, which is treated as an unstructured source, with n files is defined as $P = \{v_1, v_2, \ldots\}$, in which each v_i is a keyword that represents metadata (e.g. filename), classes, methods (procedures or functions), variables or SQL queries. Notice that C is renamed as P due to the application domain.

There are additional undirected relationships (due to memberships) between classes, methods, variables, and queries, in which a class can have methods, variables and queries, and a method can contain some variables and queries.

Chapter 3

Previous Research

The integration of information retrieval systems and DBMSs is an idea that has been around for several years, but it has been explored from an SQL perspective only a few times. This is especially true for metadata integration, federated databases, and multi-threading [113].

3.1 Data Preprocessing

Existing C search engines and libraries for information retrieval, such as the Lemur Project, Mallet, or Libbow [6, 85, 84] must create their own indexes and their own data structures for managing text files, and work completely independently from any relational database system. As stated in [61], the concept of using SQL to perform an IR search began with [23] parsing text into words using a C implementation, then storing the words in a DBMS and performing operations in the collection with indexed tables. The advantage of using an SQL implementation, as described in [56], over using existing solutions or even vendor implementations such as Microsoft SQL Server or those offered by Oracle, or “application specific” programs, is the portability of the standard SQL, its independence from the security model of the system, and fact that it does not compromise the stability and performance of the database system. Still, we accept that overall performance continues to be limited when compared with non-restrictive language implementations, and propose this approach as an alternative option for processing documents already stored in DBMSs.

Some authors [56, 52, 53] have tried extending their systems by applying the Vector Space Model without considering preprocessing or query optimizations of clustered databases, or by assuming multi-node processing or parallel computing. The OPM [83] and word stemming and parsing were developed [61], but differ from the research presented in this dissertation because they did not address completely the problem of document preprocessing within the database. Also, various query optimizations are ignored in [61] precisely because the authors used distributed or parallel computation,disguising their performance results.

This dissertation additionally explores and tailors the Dirichlet Prior Language Model for a relational database management system. The costly NOT-IN operation in the stopword removal, as well as different inverse frequency computation strategies have been ignored. Because of the limited numbers of approaches in the previous literature review, especially in the document preprocessing section, we decided to explore document preprocessing and ranking without using a computer cluster. This helped improve performance in many of these applications, and the findings presented here are supported with time experiments that will be discussed in future sections.

The rank operation is holistic. As a result, reading the whole data set is needed in order to obtain correct results. Despite this, top-k algorithms are used to avoid visiting and evaluating all of the elements in a set of lists that are combined by a given function. As stated in the previous chapter, numerous recent papers [66, 31, 25, 67, 30, 68, 31] sparked our interest in top-k algorithms. The most representative top-k algorithms are the TA algorithm [30], the NRA algorithm [31], the Rank-Join algorithm [68], and the J* algorithm [87]. The work with top-k queries gained popularity with the publication of the TA algorithm in [30], which assumed a sorted access to the data, and a later improvement required random access to the data. These first attempts at top-k queries focused mainly on ranking queries with different attributes within a single table. Since the proposal of the TA algorithm, research on top-k queries has been intense, especially with implementations and algorithms that work around the DBMS, but that are not necessarily inside the database management system, nor do they exploit the DBMS functionalities.

There have been some approaches for integrating top-k queries into an existing DBMS system. Initially, [64] presented a first attempt to implement a pipelining algorithm that worked with materialized views, showing an improvement from the traditional SQL approach since accessing every tuple in the view was not required. Later, [25] proposed a new algorithm (LPTA) for working with views, the main contribution of which was adapting the threshold value based on the linear programming approach, and proposing merging method for pipelining the algorithm. In [87], the J* top-k approach was presented, which was based on the A* to rank and manage the early joins. [68] then presented a similar algorithm to J* that also modifies the stopping condition, unlike the TA algorithm. [80] presented an approach that sends a set of queries with an application working on top of the DBMS and using a non-monotonic function. One of the contributions of the research presented in this dissertation is a comparison between an SQL implementation and a UDF implementation in our novel data layout for storing text data for working with smaller data sets. This presents a new approach to exploiting database indexes, temporary space, clustering, and the pipelining of the rank join algorithm.

3.2 Data Integration

For the second part of this research, we had to focus on the schema matching problem, which consist of finding equivalent concepts among heterogeneous sources. Although schema matching is an important research area that has been explored for quite a long time [5, 104, 108, 82, 27], schema matching between database-schema with unstructured sources has been poorly explored. In [76], the authors describe a way of using a neural network classifier of fields in two databases to discover which attributes relate to one another. Using this technique, the matches are discovered from metadata. In addition, in [86], the authors attempted to match both schemata using a set of given rules. In [9], the authors describe a system called MOMIS, which uses an Object Data Model that relies on a semantic approach to match the structured and unstructured sources. Additionally, the description of the schema is used to link the schema and the documents. Previous work has also been done on managing scientific databases. The authors in [99] presented a prototype system for managing large output files from numerical simulations. The proposed application was designed for domain experts who are not familiar with SQL, but have a scientific background. The authors proposed using intuitive searching and browsing mechanisms, from which the user can store, search, and retrieve large, distributed files from scientific simulations. In [70], an architecture for creating a collaborative centralized infrastructure for sharing scientific data is proposed. The architecture uses a search schema that allows users to locate data sets by exploiting metadata information.

Finally, in [91], we suggest a method of relating schema-keywords with terms in the documents, and generating macro and micro relationships. The matching between both sources was done using an Object Data Model, as presented in [9], and the unstructured sources were managed using XML. We extended this idea to present macro, micro, and atomic relationships, each on a different level of granularity and completeness.

3.2.1 Keyword Search

Searching keywords between a database and a collection of documents has been approached through keyword search and schema matching. In contrast to schema matching, keyword search in databases is centered on finding keywords in all the elements in the database. In a coarse view, this can be seen as generating row trees as answer sets for a given keyword. Different approaches have been developed using tuple trees, joins [51, 65, 10, 2, 79, 107, 109, 11, 75, 71, 63] or in XML databases

[8, 114, 57]. Finding links among these two sources is not directly finding schema matching or a keyword search. However, ideas from both topics have been improved upon in order to achieve this task. EROCS is one of the few attempts at obtaining a linkage between structured and unstructured data [15, 12, 11]. EROCS focuses on linking transactions and then performing data mining operations over the newly discovered links. The linkage is performed using templates for identifying entities. In [112, 74] the authors have a similar objective of linking structured and unstructured sources (and probably the closest research to the algorithms proposed in this dissertation). However, in their approach, the DBMS is analyzed as a graph and then clustered. The final result is a traditional rank on top of the nodes of a graph.

In [91, 45], we suggest a method of relating schema-keywords with keywords in the documents, and generating macro and micro relationships. The matching between both sources is done using an Object Data Model, as presented in [9], and the unstructured sources were managed using XML. We integrate the schema matching and keyword search ideas to present a more complete analysis of the existing type of links between elements and documents. We also present a novel idea on how to rank documents given these relationships within the DBMS. In addition, the research does not require a given Object Data Model for discovering these links between the structured and unstructured sources, and the linkage is performed without any entity templates. Finally, we extend the ideas presented in [40] by providing a third method for ranking the elements in both the keywords in the database and the collection of documents. In addition, a more detailed analysis is performed on the involved data structures, and the derived relationships are introduced.

3.2.2 Query Recommendation

Recently, a term "popularity" approach is explored in [72] in which each term is compared to the query terms. Similarly, the authors present the idea of naïve suggestions, but they do not explore how to efficiently compute such aggregations. Early straightforward attempts on query logs for unbiased query recommendations were presented in [33, 50]. But the authors assume previous knowledge by using these logs. OLAP has been explored less than association rules for query recommendation. However, a couple of applications have implemented OLAP to explore document collections [100, 36]. In [100] a contextualized warehouse (XML data warehouse) of the documents was built, and then queried by the usage of OLAP on relevant measures assigned to the documents. However, they focus on the metadata of the documents. In addition, OLAP sampling was explored initially in [77] as a technique for speeding up OLAP aggregations with approximate results. However, this analysis is not presented in the context of collections of keywords.

3.3 Knowledge Discovery

3.3.1 Representing Knowledge as Ontologies

The need to automate the time-consuming generation of ontologies has led to multiple solutions that rely on previous knowledge or time-consuming tasks. OntoLT [14] is a plug-in for the Protègè ontology tool. It allows the user to define a set of rules for extracting an ontology from annotated text collections. In [24], the authors present

TextOnEx (improved to Text2Onto). This prototype builds an ontology based on an NLP approach. Unlike the system presented in this dissertation, OntoLT has to rely on preconditions given by a third party, and TextOnEx requires a set of given patterns. Data mining approaches, such as [29], propose using a C4.5 decision tree to extract the main concepts. The non-leaf nodes are classes and the leaf nodes are individuals. In contrast to the approach presented here, this class extraction is performed in one pass through the data. Ontobuilder is a ontology extractor based on a schema matching approach [35]. The ontology extractor is based on heuristic methods. Doddle-OWL [34] reuses given knowledge to extract classes and relationships (mostly WordNet). It then relies on a refinement (feedback) to build a final ontology.

We realized in [37] that the query recommendation algorithm could also discover the main ontology concepts behind a particular topic. One of the contributions of the research is that classes and relations may be obtained based only on efficient computations of several discrimination values (correlation and lift.) Unlike the proposed solutions, this requires minimal human intervention and the tuning of only a few parameters.

3.3.2 Deriving Cubes from Ontologies

Using OLAP for summarizing text corpora has been presented in several publications [73, 78, 116]. Early in [73], the authors presented an approach to loading all the documents inside a DBMS. The documents were all loaded into a star schema that allowed the users to utilize traditional OLAP algorithms. The computations were focused on keyword frequency. Loading all of the data into a predefined schema that will allow exploration using traditional OLAP techniques represents a major drawback of this early works. This approach is prohibitive when dealing with a data set containing a large number of dimensions, unlike the vertical layout approach which allows the storage of large dimensional data sets. In [78], the authors propose Text Cube. This approach focuses on computing a partial cube by using only a partial materialization. The algorithms in [78] are focused on obtaining "on-demand" OLAP queries with the optimal processing cost, using a greedy algorithm. This is unlike the approach presented here, in which the focus is on computing all of the existing dimensions of the data cube at once by taking advantage of the text sparsity. More recently in [116], Topic Cube is proposed to obtain a more complex analysis than just data summarization. The purpose of this new OLAP model is to extract dimensions for existing text data using the probabilistic latent semantic analysis. Despite the fact that the model and results are interesting for generating a possible ontology, the authors do not to propose a new data structure for the multidimensional data cube (they compute the SQL queries as required), but focus on the quality of the built hierarchy.

In [37] and [38], we were able to adapt a traditional OLAP data cube to efficiently process a sparse vertical fact table. However, we were only computing a set of dimensions, and the scalability of the algorithm was limited to only a few dimensions. In addition, the papers were focused on the aggregation of several measurements inside text corpora to produce the most frequent cuboids for generating an unsupervised ontology. This ontology was the result of a post-processing phase on the resulting cuboids. In this dissertation, a given ontology is assumed, and it is used to summarize a set of documents. Thus, one of the contributions of this OLAP approach focus on a new algorithm that integrates the hierarchy given by an ontology into a single data structure called CUBO. In addition, a formalization and study in depth of the theoretical and experimental complexity of our algorithm are presented in this dissertation. Finally, the proposed algorithm considers a way to manage several types of hierarchies that can be present in an ontology.

3.3.3 Integration and Querying of Programs and Schemas

Control and data dependency analysis of programs have been explored for code maintainability, reverse engineering, and compilers. Despite this, the interaction between source code and a database's schema has rarely been explored. In [60] the authors propose the DB-MAIN case tool. This tool automatically extracts data structures and control and data dependencies declared in the source code (including all the explicit and implicit structures and constraints). The result of this tool is a conceptual representation of the data structures and the relationships between them. DB-MAIN proposes three techniques for capturing the dependencies in a program which are based on applying heuristics after a pattern-matching analysis. Unlike the previous approaches, the proposed approach is based on finding approximate matches. Moreover, the proposed approach allows the flexibility of analyzing a larger variety of languages and pieces of code. DB-MAIN supports only a few languages (such as COBOL). Furthermore, a DBMS is relied upon to perform the extraction and exploration of the matches. In the keyword search domain, several algorithms have been proposed for performing efficient searches [17]. From this family of algorithms, the closest similarity to the research presented in this dissertation is found to be in the DBXplorer system [2], BANKS [10], and DISCOVER [65]. The newly proposed graph algorithm is partially based on the DISCOVER algorithm in the exploration style. However, in DBXplorer, DISCOVER, and BANKS, the challenge is to find those tables and attributes that are related to PK/FK constraints. This differs from our algorithms, which focus on exploring the database's schema and not on the data within. Finally, our proposal is the result of ongoing research in the field of data integration and joint exploration between structured and unstructured sources. This approach was originally implemented to integrate semistructured data with structured data, as presented in [91], in which the basic notion of a "link" was introduced. In addition, several algorithms for finding those links were introduced. Later on, these ideas were expanded upon to perform efficient approximate keyword matching inside the DBMS [40]. In this dissertation, the ideas presented in [90] for preparing a data set for data mining and the ideas in [37] for OLAP exploration are extended to allow complex analysis of the resulting matches. Finally, this dissertation formalizes the notation in [42] and introduces efficient ranking methods for the obtained references and source code files.

Chapter 4

Processing Documents

Document storage, scoring functions, top-k queries, and the discovery of relationships require different optimizations and algorithms tailored to the DBMS. We separate the general IR process into two phases: the storage phase and the retrieval phase (see Figure 4.1). Document storage requires efficient document preprocessing and term storage. Document retrieval requires effective aggregations and top-k algorithms. Without loss of generality, the words a "term" or "keyword" are use interchangeably in this chapter.

4.1 Storage

We separate the general IR process into two phases: the storage phase and the retrieval phase. The flow of the operations for the three ranking models is related to the desired optimizations or selected parameters from the user. The three models

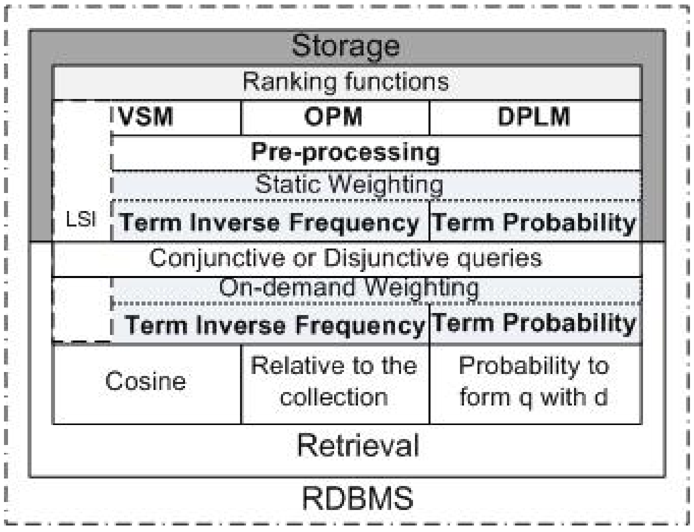


Figure 4.1: Storing and Ranking Framework.

have a uniform preprocessing phase, then the term weighting step varies depending upon the retrieval model. Later, the retrieval phase may contain a weighting task needed for the scoring of the ranking model. The last step is the scoring of the ranking model.

The document preprocessing step takes every document and creates a descriptor vector as no hierarchical model (or "bag-of-words"). The vector is the result of removing stopwords and applying the Porter Algorithm [98] to every non-stopword. This algorithm has been adopted regularly as the default choice for stemming, although there exist other options such as Lovins, Dawson, and Paice, among others [98]. The main purpose of using a stemming approach is to lower the number of dimensions (words) in a document and still be capable of "describing" the document using a reduced descriptor vector while losing as little information as possible.

4.1.1 Data Layout and Indexing

The data layout defines how the document data are stored in the DBMS. Such a layout is important for speeding up the retrieval models and also allowing that various techniques can function. Storing the document collection requires a data layout that will allow pipelining for the top-k algorithms, indexing and clustering of documents, and terms for weighting and retrieval. Later on, it will be required also to allow the generation of the document-schema relationship discovery.

We use the *Baseline Inverted File* technique for weighting and indexing the terms [117]. This technique consists of two principal structures. The first of these is a search structure (vocabulary or dictionary) that stores every distinct term in the collection, and a pointer, or a pair of primary and foreign keys in the relational model, in a corresponding inverted list. The second structure is the inverted list structure in which each entry stores terms and the associated set of frequencies of the terms in a document [79]. This results in the storage of a descriptor vector of every document in a huge matrix of different numbers of terms and N number of documents in the collection. The main advantage of using an Inverted Files technique relies on the fact that the frequency of every term has already been stored. This frequency can then be easily manipulated into several ranking strategies. Storing vectors in the relational model is difficult, and we will assume a vector to be a single column relationship for the relational model.

The schema in the database (see Figure 4.2) includes a table "stopword" which contains the collection of indexed undesired words. A table "document" stores the

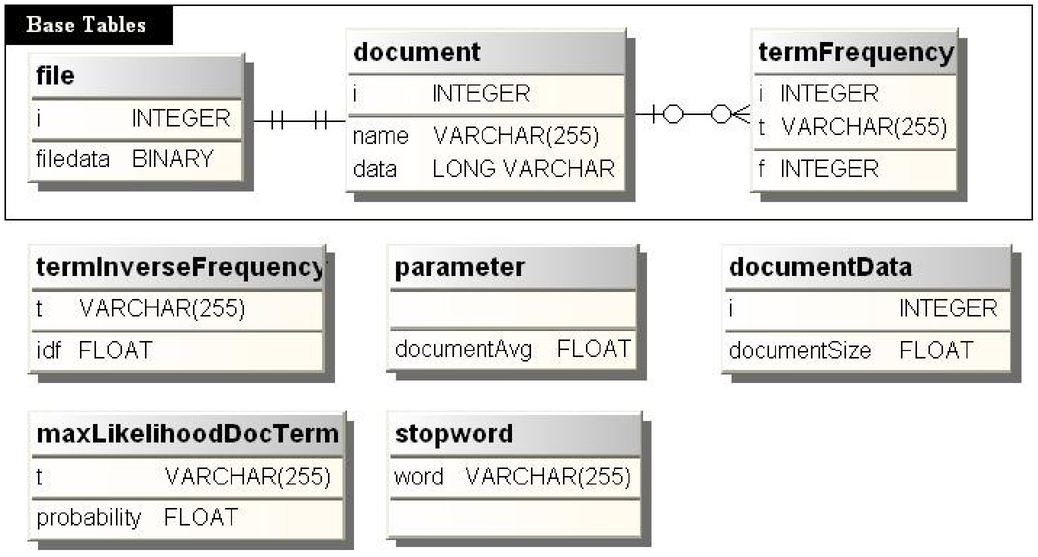


Figure 4.2: Vertical Layout Storage

metadata of the document, indexed by the document's unique identifier i selected as the primary key. The table "parameter" includes the corpus statistics as a requirement for some ranking strategies. The table "termFrequency" contains the terms of each document and the term's frequency. The "termFrequency" table is indexed by t and i, and the terms are clustered by i, because ordinarily they are retrieved together to be compared in any of the chosen ranking models. For OPM and VSM models, the weight of each term in the collection is computed with an inverse frequency table "termInverseFrequency," indexed by term. In the Dirichlet Prior Language Model, the probability of each term appears in the "maxLikelihoodDocTerm" table. For OPM and DPLM ranking strategies, data was collected data from the documents and those values were stored in "documentData."

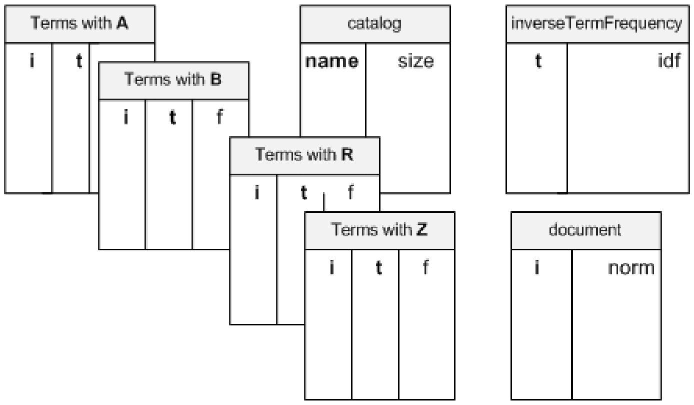


Figure 4.3: Bucket Data Layout.

4.1.1.1 Data Layout for the Top-k Algorithms

The top-k computation required designing how to store the documents in a relational database system in order to allow pipelining and enhance retrieval performance. But storing the data layout to fit inside a DBMS is not a trivial task, mostly because the data must be arranged in a way that will allow the database system to conduct a fast retrieval without performing full table scans in each table of the collection for each search. Therefore, we decided to explore with a layout different from just using a single huge table containing all the document-term pairs in the collection (see Figure 4.2). We originally started with a huge document-term table that stored all the document-term-frequency entries of the collection, making it difficult to manage, and not allowing any advantage when pipelining. We then decided to change the data layout in order to base it on a divide-and-conquer strategy. so parts of the document-term-frequency entries were stored in different tables based on some term interval (see Figure 4.3).

Support summary tables were stored for the ranking model that contained the norm of the documents and weight of the terms, and document-term-frequency entries were divided into ranges. This results in a collection of tables containing just a "bucket" filled with some terms of a document, and was done precisely to exploit the sparse nature of documents, and due to the fact that the number of terms to be searched is generally small compared to the total size of the collection. Therefore, a physically close storage of all the related terms in a table was enforced. A clustered index was also added for grouping all the related terms of the same document in a table based on an index (to allow faster retrieval), allowing us to guarantee that all of the related terms of a document are stored next to each other (e.g. {doc1,house}, {doc1,home}), since related terms may be needed that are in the same bucket but are part of the same document and may be aggregated together if needed during the "scoring phase." Additionally, some summary tables were created, in which we stored the norm of each document, and in another table, the term's *idf* value was stored. When m tables are required for ranking, user temporary tables are created, as will be explained later for some techniques. In some cases, these temporary tables will be indexed by a surrogate key, from one may simulate random access to the ordered records in the table. This technique is similar to the data layout presented in [25].

4.1.2 Document Preprocessing

For the document splitting phase, a recursive function was proposed [88] for portability and for performing the splitting task (see Figure 4.4), which normally has been solved using external implementations, language, DBMS specific implementations,

```
WITH TERM( i, word, g, k , j ) AS
( SELECT i, word, g, k, j FROM T
  UNION ALL
  SELECT TERM.i+1
   ,CAST(SUBSTRING(T.word,TERM.i-TERM.k
                  ,TERM.k+1) AS text)
   ,CASE WHEN SUBSTRING(T.word, TERM.i,1 ) = ' '
         THEN TERM.g+1
         ELSE CASE
           WHEN TERM.g=1 THEN TERM.g+1
             ELSE TERM.g END
         END
   ,CASE WHEN SUBSTRING(T.word, TERM.i,1 ) = ' '
         THEN 0 ELSE CASE WHEN TERM.g=1
         THEN 1 ELSE TERM.k+1 END END
   , 1
   FROM T
   INNER JOIN TERM
   ON ( TERM.j = T.j )
   WHERE TERM.i < DATALENGTH(T.word)+2
)
INSERT INTO RESULT
SELECT i, word, g, k, j
FROM TERM;
```

Figure 4.4: Recursive Splitting Function.

or n-gram parsing [61]. We started with the initial text to be parsed in a temporary table. Using "term" as the table that makes the recursion possible, and then we select the cumulative word before the "word splitter." In this case, we decided to use a blank space as the separator, but this could be replaced with any other character. The final result was then stored in the temporary term table for future use.

4.1.3 Stopword Removal

Upon the conclusion of the splitting step, the resulting words in the table were stemmed. Prior to this, however, the stopwords were removed, as clearly described in [61]. These tasks have been executed normally with a NOT-IN operator. Based on previous research, we decided to modify the stopword removal query by selecting all the non-matching values in a LEFT-JOIN query. This optimization was explored in [93], and has not been presented as an alternative technique for comparing and eliminating stopwords in the original document.

```
SELECT keyword AS term
FROM Splitter('{Query String}')
LEFT JOIN stopword
ON word = keyword
WHERE word is NULL;
```

4.2 Exploration of Documents

The preselection of the documents to be ranked can be achieved for conjunctive or disjunctive queries, as well. The idea is to reduce the number of terms and documents to work with in the on-demand weighting and ranking models. Querying a collection for disjunctive or conjunctive operations traditionally requires building a query dynamically for searching for terms in the frequency table. Then, these selections are constructed by adding “AND” or “OR” operations at the end of a selection. With this optimization, the previous techniques for preprocessing a query are reused, by purging the user query and then splitting and storing it into the query temporary table. In a disjunctive query, the selected documents are the result of a LEFT-JOIN optimization.

```
SELECT DISTINCT i
FROM termFrequency tf
LEFT JOIN Splitter('{Query String}') pa
ON PorterAlgorithm(pa.keyword) = tf.t;
```

The conjunctive query corresponds to a division relational operation in the database, including the “termFrequency” table and the query table, necessary for finding all of the documents that contain all of the query terms. Thus, we execute an optimization discussed previously in [93], is executed, wherein this division operation is expected by adding a GROUP-BY, in order to remove duplicates and to count how many unique terms each document contains from the query. Finally, in a temporary table stores the number of words in the query. As part of the implementation, a single occurrence of a term in a user query is assumed.

```
SELECT i
FROM termFrequency tf
LEFT JOIN Splitter('{Query String}') pa
ON PorterAlgorithm(pa.keyword) = tf.t
WHERE pa.keyword IS NOT NULL
GROUP BY i HAVING COUNT(*) = {# Query Words};
```

The retrieval functions for the three models imply a sequence of natural joins with the termFrequency and the "weight" table. The objective when designing both implementations was to reduce the size of the term tables as quickly as possible. Thus, several temporary tables were created, such as the table "doc," which contains the preselected documents from the disjunctive or conjunctive query, but represents the set of all the documents that contain the desired terms. This temporary table is used as part of the on-demand technique for verifying whether or not the termInverseFrequency table contains all of the required terms. Finally, it is assumed that the norm of the query has already been precomputed.

$$\begin{aligned}
&\rho_t(doc \bowtie termFrequency)\\
&\rho_{tmpd}({}_i\mathcal{F}_{sum(qd)}((t \bowtie Query)\\
&\bowtie termInverseFrequency))\\
&\rho_{tmpqd}({}_i\mathcal{F}_{sum(d)}(t \bowtie termInverseFrequency))\\
&\Pi_{i,qd/(d*q)}(tmpd \bowtie tmpqd) \qquad (4.1)
\end{aligned}$$

In order to include our SQL queries, we present them in relational algebra (instead of a larger SQL). For the Vector Space Model (see Equation 4.1), a series of aggregations in the "tmpd" and the "tmpqd" temporary tables is computed. The "tmpd" table represents the norm of the documents that contain the desired terms and the "tmpqd" table stores the dot product between the query table and each document. At the end of the execution of the queries, the three tables are joined to obtain the score of each document without needing to perform an aggregate function in this operation.

The ranking function involves several joins, including the term frequency table and the inverse frequency table. The first part of the query computes a join between the “query” table and the inverse frequency table, and this query is computed once. Then, the rest of the query computes the norm of the term vector of every selected document, which includes a join operation between the frequency table of the terms involved in the operation and the inverse frequency table. Then, using the selected documents, the norm of each document is computed with an inner join between the term frequency table and the inverse frequency table. The last part of the query is a join with the selected document’s norm and the “tmpqd” table and the already computed “query” table norm. The ranking function includes five joins for obtaining those results, but some tables are just obtained once and reused (i.e. the doc table). Also, as explained previously, in the static frequency table technique, the inverted frequency table is already precomputed. However, in the on-demand technique, every term that is needed for its frequency must be searched in the term frequency table. If the term is found in the term frequency table but not in the inverse frequency table, then its term inverse frequency must be inserted into the table. The “termFrequency” table is stored in a cluster based on the document ID, keeping the terms in the descriptor vector together. This is done in order to obtain better performance when implemented in the DBMS.

The Okapi Probabilistic Model (see Equation 4.2) requires more information than the Vector Space Model, since it requires the length of the document (considered without stopwords) and the average size of a document in the collection.

$$\begin{aligned} &\rho_t(doc \bowtie termFrequency) \\ &{}_{i}\mathcal{F}_{sum(rank)}(termInverseFrequency \\ &\bowtie (documentData \bowtie (Query \bowtie t))) \end{aligned} \tag{4.2}$$

The implementation of the Okapi formula in SQL is performed with three inner joins. The deepest query involves the matching of documents of the desired queries, as in the Vector Space Model. Then another inner join is performed with this temporary document table and the termFrequency table to obtain the second and the third parts of the formula. This leaves a last inner join operation, in which each document term is multiplied by its inverse frequency, and then an aggregation operation is applied on document ID with a SUM on the final score of each term and a GROUP-BY.

The Dirichlet Prior Language Method (see Equation 4.3) is very close to the implementation in SQL of the OPM, since it performs joins in similar sized-tables. Despite this, the ranking elements and the results of the aggregation are very different, due to the weight term product, the term weighting, and the ranking equation.

$$\begin{aligned} &\rho_t(doc \bowtie termFrequency) \\ &{}_{i}\mathcal{F}_{product(rank)}(maxLikelihoodTermDoc \\ &\bowtie (documentData \bowtie (Query \bowtie t))) \end{aligned} \tag{4.3}$$

The DPLM implementation is performed very similarly to the Okapi Probabilistic Function, but instead of performing a SUM in the aggregation, a product is obtained, to which the given formula is applied. The overall execution time is very similar to the OPM.

A popular alternative for ranking is the Latent Semantic Indexing approach. Latent Semantic Indexing relies on the premise that using direct match on keywords (as with our previous techniques) may give incorrect results [55]. As a result, LSI reduces the number of keywords in a document in order to have a new representation of the documents in a difference space, where all the documents have the same vector size. Despite this, latent semantics can also be exploited using our described data layout with some precomputations that are the result of Singular Value Decomposition (SVD).

The process begins with a matrix A, that corresponds to the document-term matrix. The A matrix is of size C dimensions and the different numbers of t. Each cell in matrix A represents the frequency of that keyword in each document. This matrix is decomposed using the correlation matrix, householder decomposition, and QR factorization into $U\Sigma V^T$, such that Matrix Σ are the singular values. The singular values are then selected by magnitude and reduced by selecting the most important k values (the remaining values are set to 0). These singular values are what we consider to be the latent semantic. Finally, using these new matrices, the query needs to be transported into the same k-th space by using $q^T U_k {\Sigma_k}^{-1}$. On the other hand, the transformation of the documents is contained in V_k. The size of those support tables in the data layout will be the following: matrix V is of size k by N, matrix Σ is of size k by N and matrix U is of size N by the number of different terms in the collection. Once this mapping of the query and the documents has finished, the new obtained vectors with k elements in each need to be ranked. To rank them, the Vector Space Model is applied to these vectors to obtain the final score. The computation in SQL is reduced to a set of precomputations and temporary matrices that are needed for obtaining the latent semantic. The rest of the operations are reduced to the VSM.

4.3 Top-k Rank-Join

The Rank-Join algorithm, described in [68], is based on the TA and NRA algorithms. The authors propose that the Rank-Join Algorithm can be extended as a new SQL operation that can be used in a query plan (pipelined). As explained in [68], this algorithm exploits the maximum and minimum values of an attribute in two sorted lists, in order to obtain a computed threshold (or cutoff value like in TA), and it only reports the documents that are above this computation. The Rank-Join algorithm, like most top-k algorithms, requires a monotonic increasing function in order to guarantee that it can find a threshold, and that no value smaller than this will have a higher score. In order to obtain a monotonic increasing function in the ranking with the Vector Space Model, the top-k is performed after each term has been associated with the query vector. Therefore, the sorted lists contain weighted values associated with the query and the remaining operation is an addition, which can be guaranteed as a monotonic increasing function.

As explained in [68], the Rank-Join algorithm is different from the NRA and TA algorithms because it does not have a restrictive data access pattern. Instead, the algorithm only keeps the scores of fully completed join combinations, allowing the algorithm to be independent of the order in which the data are accessed. In contrast, an algorithm such as NRA makes assumptions regarding partially viewed join combinations and is thus more dependent on the access pattern. The Rank-Join algorithm guarantees obtaining the top-k items within two ordered lists, but requires a further scheduling extension for pipelining. A summarized version of the Rank-Join algorithm as presented in [68] is described in Figure 4.5.

```
Data: Rank-Join($L_1, L_2, \ldots, L_M$)
Result: Top-k documents
1 Generate a valid join combination;
2 For each resulting join combination compute the score;
3 Let $(t_l)^{max}$ be the top score in List $l$. Let $t^{last-seen}$ be the last seen score in List
  $l$. Let $T$ the maximum of the following $m$ values $\{F(t_1^{last-seen}, {t_2}^{max}, ..., {t_m}^{max})$,
  $F({t_1}^{max}, t_2^{last-seen}, ..., {t_m}^{max}), \cdots, F({t_1}^{max}, {t_2}^{max}, \ldots, t_m^{last-seen})\}$;
4 Let $L_k$ be a list of the $k$ join results with the maximum combined score seen
  so far and let the $score_k$ be the lowest score in $L_k$; halt when $score_k \geq T$;
```

Figure 4.5: Rank-Join Algorithm

Unfortunately, there is a problem with the Rank-Join algorithm, due to the fact that pipelining requires the modification of the number of "documents" to be retrieved as one traverses deeper into the query plan.

In order to speed up the document retrieval, we experimented with top-k algorithms using the VSM. Traditional ranking in SQL takes every row into consideration and obtains the final score for all the joins in J. It then takes the top k documents for the whole ranked and sorted data set. The main advantage of this type of query over the rest of the approaches is that it delegates the tasks of deciding which is the best approach for joining the tables by using the query optimizer of the DBMS. This could represent a great advantage when there are large tables and small tables present together within the collection. Also, early tuple pruning could reduce the total time required for returning the top-k documents, which is not possible if there is not a smart scheduling technique to support the Rank-Join algorithm.

```
SELECT TOP 5 i, SUM(tbl.rank)
FROM ( <union of m tables> ) tbl
GROUP BY tbl.i
HAVING COUNT(*) = (SELECT COUNT(*)
FROM irtopk_term WHERE t='APPLIC' OR t = 'BECOM')
ORDER BY SUM(tbl.rank) DESC;
```

The query above represents the traditional SQL ranking, from which "<union of m tables>" represents a couple of joins which obtain the terms' frequencies and weights for the ranking formula and the norm of each document. The norm of the query vector is considered to be precomputed when computing this query.

The SQL approach (SSQL) had to be tailored in a different way than the traditional ranking. Instead of using a single query, we implemented a variation of the Rank-Join algorithm proposed in [25]. We initially decided to use temporary tables instead of views to speed up the execution, but the time difference from the actual materialization of the results was negligible. Temporary tables also relied on the stability of the in-database connection in the used DBMS, which resulted in connection problems with larger datasets when the connection had to be kept alive during longer periods of time. This made the connection subject unexpected restarts that avoided the reuse of temporary tables. Due to this, we relied on indexed tables to speed up the execution of predicates and the materialization of tables to guarantee intermediate results reuse.

Temporary tables were used to hold partial result datasets with immediate reuse, mainly because temporary tables do not require to be stored in the persistent objects catalog of permanent tables, and they have the capability of adding indexes, as well. A surrogate key had to be included for allowing sorting, in order to simulate random access to each sorted table. Once the sorted tables were obtained, each element was accessed individually, as described in the modified Rank-Join algorithm (see Figure 4.6). The access of both lists is applied evenly. In other words, there is no particular priority to access a row first from a specific table. This same access pattern is followed in the rest of the algorithms presented here.

The Rank-Join User Defined Function (UDF) was coded as a table value function (TVF) because of the flexibility that exists in opening connections inside the DBMS, managing the data, and returning tables as a result of a function inside the DBMS. Performance is gained mainly when in-database extensions are used instead of coding the application outside the DBMS. The UDF's capability of working in-main memory is exploited, in contrast to an external application that requires accessing the DBMS through a connection interface (e.g. JDBC).

Inside the UDF, we had to read both tables, compute their scoring value, and

```
Data: SQLRankJoinAlgorithm($L, R, k$)
Result: Top-k documents
1 Retrieve and sort $L$ and $R$ lists based on the resulting score;
2 Insert both sorted lists $L$ and $R$ into a new lists $\hat{L}$ and $\hat{R}$ using a a surrogate
  key to simulate a random access, and apply an index on the surrogate key and
  the document id;
3 Iterate through the lists evenly, and compute the new $T$ for the new visited
  values. If there is any new valid join, add into a valid join table, preserve the
  sorting in such table and retrieve all the needed elements having higher or
  equal score than $T$. Perform each operation on the indexed tables, using the
  surrogate key as a pointer to the end of the list and the document id to hash
  the result;
4 If the stopping condition for the Rank-Join has been reached, return. Else if
  there are still tuples to retrieve goto step 3, else take the top needed values to
  return the top $k$;
```

Figure 4.6: SQL Rank-Join Algorithm.

sort them. Then, row by row reads were processed from both lists following the Rank-Join algorithm, and the execution was suspended when the stopping condition had been met. The main difference between the SQL implementation and the UDF is in Step 3, in which it is possible to work without a surrogate key, but the capability of exploiting the indexes created in the sorted table in the DBMS is also lost. Most of the computations are executed in main memory and stored in a temporary array holding all the join results (sorted by size). This Rank-Join algorithm is mainly based on a sorted list which manages the valid J results.

We took advantage of the 2 GB memory space (specific to SQL Server) that can be allocated inside the DBMS where all of these computations were performed in main memory, speeding up the execution of the TVF. The UDF call in SQL needs both table names as arguments, the norm of the query vector, the terms to look for in the given tables, and the number of results to return. The TVF has limitations depending on the DBMS and the allocation of space for UDFs. UDF Rank-Join, as well the SQL implementation, accesses both lists at the same time. The function can be called as follows:

```
SELECT *
FROM
RANKJOIN('irtopk_A','irtopk_B', 5, 'APPLIC','BECOM',5)
```

An improved version of the Rank-Join algorithm, described in [69], implies managing a hash-join data structure for each visited tuple for each list, and a priority queue for sorting the valid join tuples. The improved UDF-TVF implementation exploits both data structures in main memory. These data structures approach the main issues in the list-based Rank-Join algorithm. By using a priority queue for holding the valid join tuples, it is possible to improve the bottleneck of managing these tuples. At the same time, the access to specific joins for finding a faster joins is achieved by hashing the data tables. This algorithm is similar to our SQL implementation, due to the fact that the hash access can be achieved in SQL when storing the tables. Unfortunately, the sorted storage remains as the main limitation for the SQL implementation and here this problem is addressed here by using an implementation in a UDF.

The pipelining strategy of the UDFs and SQL Rank-Join was performed following a naïve scheduling strategy in both implementations. We first grouped all the terms that were part of the same bucket (e.g. 'ARCHITECTUR' 'ASSOCI') and ordered the table scheduling list in alphabetical order. The largest table size in the involved tables was obtained and reduced by a constant $\alpha = 30\%$ of each level in the pipelining. Hence, the $\alpha(MAX(|L_1|, |L_2|))$ join result of the first two tables was obtained. We continued reusing the resulting table of the Rank-Join algorithm and computing the new k for each level, adding new tables from the scheduling list until the work was completed. The join results are then the top-k results of all of the desired tables. This pipelining technique will not work with disjunctive queries, because if some documents are discarded early by lower ranked terms, the result will have a lower accuracy than the actual top-k documents.

4.4 Experimental Validation

The experiments presented here were designed to test the performance and feasibility of each step of the research. The first part of our research focused on testing the bottlenecks and performance of the algorithms in document storing and ranking. The second section was designed to verify the performance and accuracy of using a top-k approach. The last section of the experiments focused on performance of the relationship-discovery process and the number of relationships that were found in the given data set.

The experiment were conducted on a computer with an Intel Core Duo P8700 processor at 2.53 GHz. The system hardware configuration also had 4 GB of RAM and a Western Digital WD2500 hard drive with 250 GB of space. The software configuration was running a Microsoft SQL Server Developer Edition (32-bit), version

9.00.4035.00. All of the data sets were loaded in such a DBMS engine, and a different database was created for each section of the experiments: document storage and retrieval, top-k retrieval and relationship discovery.

4.4.1 Document Storage and Retrieval

The collection of documents for testing the document preprocessing, storage, and retrieval was prepared using the publicly available DBLP bibliography and ACM Digital Library abstracts, using an automated program (crawler) for preparing the corpus (see Table 4.1). In some cases, the abstracts were larger than expected and were reduced to less than one-thousand characters. In order to proceed with the experimental section, we decided to work with two collections. The first was a smaller collection from a corpus larger than 18000 documents, creating test runs of 500, 1000, 2000, 4000, 8000, and 18442 sets of documents. The second was a larger collection, in which the original corpus was repeated 10 times for testing the search models with more than 180000 documents. For the rest of the tests that the documents, abstracts from the test bed were assumed to be already loaded into the database. Then these strings were preprocessed and stored them using the Baseline Inverted File technique, as presented in the entity relationship model.

Table 4.1: Documents in the Collection.

Docs	Dif. Terms	Word Avg	Term Avg	Std dev
18442	782421	98.43	55.7	24.94

The selected test workload was designed with five terms based on the maximum, average, and minimum frequencies. The selected keywords for performing the keyword-searches were: design, simulation, island, marginal, and zurich, for performing conjunctive and disjunctive queries (see Table 4.2). The workload was designed for querying all of the possible combinations with any of these terms, in order to measure the performance of querying a term in the collection, although the combination with all of the terms was ignored for measurement purposes. In these experiments, all of the retrieved documents were considered relevant, and for the time being, the precision and recall measurements were ignored, while focusing the experiments on performance. All of the measurements in the experimental part are given in seconds when not specified.

Table 4.2: Searched Terms.

Keywords	Frequency
design	8914
simulation	1442
island	17
marginal	16
zurich	1

The reviewed the performance of our implementation for document preprocessing was reviewed with a recursive query, and the results (see Table 4.4) showed that the performance for splitting documents is approximately 3690 words per second, on average. The implementation can be extended to work in parallel for a much better ratio, as shown in similar research by [56, 52]. At the same time, the degradation of using an SQL-based approach was compared to a UDF implementation. As shown in Table 4.3, the UDF clearly outperforms the SQL implementation, although the recursive query is valuable for the portability that is given by being built using ANSI SQL.

Table 4.3: Document Splitting (UDF vs Recursive Queries) in Seconds.

Documents	Avg. Words	UDF	Recursive Query
2500	31	0.1	19
5000	32	0.2	39
7500	32	0.4	59
10000	32	0.5	79
12500	32	0.7	107

The optimization analyzed in the storage phase corresponds to the NOT-IN and LEFT-JOIN operands for eliminating the stopwords from the document temporary table. The results showed a reduction of 1-6% in the total time required to remove the words and store them in the frequency table (see Table 4.4). The LEFT-JOIN optimization always proved to be faster than using a NOT-IN operation, but the improvement percentage depends on the number of elements in both tables. However, these optimizations, the splitting and the removal of stopwords, represent more than 80% of the total storage time (see Table 4.5).

Table 4.4: Stop-word Removal & Splitting in Seconds.

Docs	NOT-IN	LEFT-JOIN	Processing total time
500	11	10	14
1000	22	20	30
2000	46	43	62
4000	93	88	130
8000	195	184	274
18442	430	402	578

For the term weighting step, we analyzed the on-demand and static techniques for computing the inverse frequency, or probability, of the desired terms. Surprising results (see Table 4.6) were computed for the Okapi inverse frequency, including a high overhead when verifying the existence of the required terms in the partial

Table 4.5: Storage Time for On-demand in Seconds.

Docs	LEFT-JOIN %	Preprocesing %	Total time
500	37	52	27
1000	34	52	58
2000	35	50	123
4000	34	50	259
8000	34	51	539
18442	35	50	1159

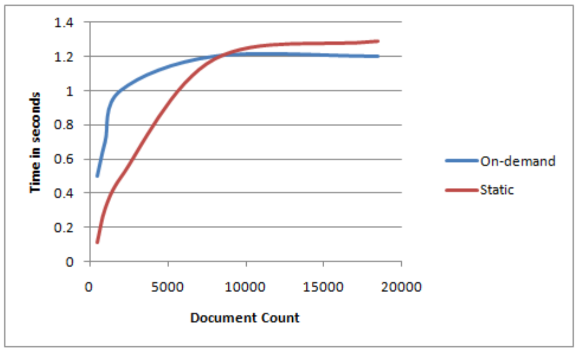


Figure 4.7: Inverse Frequency Computation for Workload.

inverse frequency table. Thus, the time for executing this background check and then inserting the required terms is higher than weighting the whole collection in one pass.

At some point, (see Figure 4.7) if no new terms are searched, the time consumption of the on-demand technique will remain with just the overhead cost of the LEFT-JOIN operation searching for missing terms in the partial-term weight table. The performance remains steady when the total number of different terms has already been reached. On the other hand, time consumption of the static inverse frequency or probability computation will surpass that of the on-demand technique as the number of terms keeps increasing. Thus, deciding which strategy to apply depends on the workload and the number of different terms that are part of the collection.

Table 4.6: Weighting Techniques for Okapi in Seconds.

Documents	On-demand	Static
500	<1	<1
1000	1	<1
2000	1	<1
4000	1	1
8000	1	1
18442	1	1

4.4.1.1 Retrieval

The document preselection on our three ranking models was tested. The performance with conjunctive queries is faster than the performance with disjunctive queries for the premature reduction of the number of terms. The search for documents that contain all the searched terms is a time-consuming operation. However, the limited number of findings speeds up the overall execution of a conjunctive query, except in the first case, in which obtaining all of the documents that contain all of the queries proves to be a more time consuming operation than the disjunctive query. After more keywords are added to the query, the disjunctive operation involves a larger number of matches, and therefore, a slower performance. According to the results that we obtained (see Table 4.7), the VSM implementation is slower than OPM and the DPLM implementations, except in the first case, when just one keyword is involved in the search. In general, due to the number of operations that the VSM must perform, Okapi and DPLM appear as much better options. Also, it is well-known that Okapi is a good model for fixed-length documents, and due to the similarity amongst the join operations between OPM and DPLM, a similar behavior is observed in those algorithms, as well. Also, the conjunctive query by itself is the most time consuming operation (GROUP-BY) in the scoring, but this early implementation in the document selection retrieves far fewer documents than the disjunctive operation. Thus, in the overall time for the ranking operations, the conjunctive operation appears as a much faster option.

Table 4.7: Conjunctive (C) & Disjunctive (D) Queries (time in seconds).

180000 documents						
	VSM		OPM		DPLM	
Terms	C	D	C	D	C	D
1	6	7	12	4	6	6
2	4	15	2	4	2	7
3	4	24	2	4	2	11
4	4	34	2	4	2	13

We consider these results promising for future optimizations, and they have proven to reduce the SQL overhead that has been assumed as one of the principal problems when working with a declarative language such as SQL. Also, these experiments aided in the exploration and building of an application with this SQL alternative, taking advantage of indexes, clustered storage of related terms, user-defined functions, and faster SQL operations. The results obtained showed under which circumstances any of these optimizations can be applied, taking into consideration the nature of the collections and the workloads.

In particular, the recursive query, described as part of the preprocessing tasks of the documents, proved to be a very effective option for breaking up the documents and storing them, pivoted and purged, in a temporary table in a standard SQL implementation. Contrary to previous research, the solution presented here is not attached to the DBMS proprietary functions, nor is it suited for extracting n-grams. The LEFT-JOIN operation gives an improvement of 1-6% instead of using a "traditionally proposed" NOT-IN operation. This saves a lot of time in the overall process of loading documents. Document splitting and stopword removal have to be executed for every document, parsed and stored into the collection. The same idea was also used as part of the on-demand term weighting step, reducing the total overhead of verifying the existence of terms in the inverse frequency or probability table.

For the static and on-demand techniques, we observed that a static ranking will normally be a much faster approach, but when it is known that the collection includes millions of different terms, and the queries are over a few of these terms, all of the terms in the collection do not need to be ranked. We also observed that for values greater than 10000 different terms, the term weighting becomes the most time consuming operation in the overall process of preprocessing, storing, and ranking. Finally, the fact that previously studied query optimizations can be used and applied in the context of information retrieval to minimize the limitations of a database as a search engine is an idea that has not been previously explored, as discussed in the related work section. Also, we consider the improvement found to be significant, especially when taking into consideration that these operations run for thousands of documents and terms in the database system.

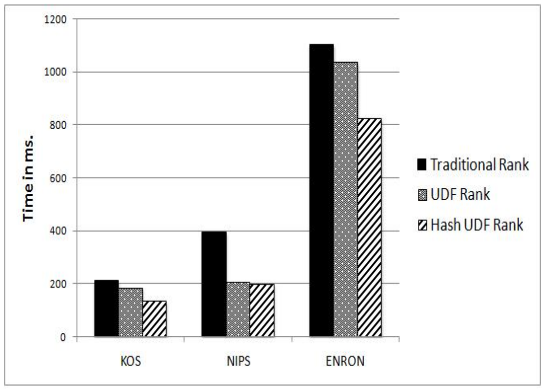
Table 4.8: Top-k Test Data Sets.

Data Set	Docs.	Terms	Entries
KOS	3430	6087	353157
Nips	1500	10950	746313
ENRON Email	39861	24678	3710417
NYTIMES	299752	7979	69679424

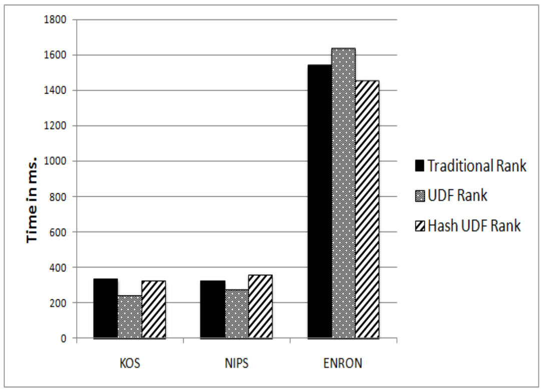
4.4.2 Rank-Join

The driver code and the UDF were implemented using C#, in which the UDF was developed as a Table-Valued Function (TVF). Different-sized data sets were used to test the Rank-Join implementations in SQL and UDFs, and the traditional SQL ranking (as the benchmark). The "Bag of Words Data Set," taken from the UCI Data Mining Repository [4] (see Table 4.8), each data was identified by its number of document-term entries in the plots. In this experimental section, the performances of the SQL Rank-Join algorithm, the UDF Rank-Join algorithm, the Hash Rank-Join algorithm and the traditional ranking algorithm were compared using the first three data sets, and the last collection was used only to test accuracy of the top-k elements due to its non-uniform distribution.

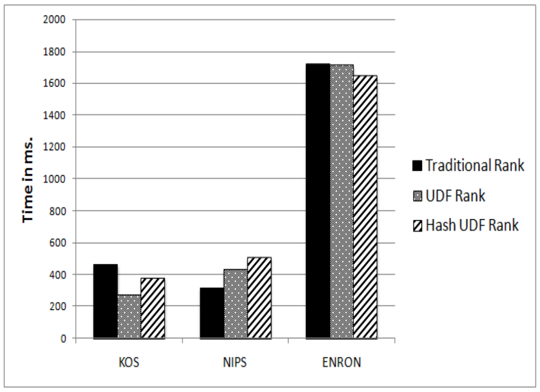
The experiments were focused on performance and accuracy in the collections. First, we compared the SQL approach and the traditional SQL top-k retrieving the top 5, with two tables involved for each collection. Then, we compared the rest of the UDF Rank-Join implementations were compared with the traditional top-k search. The first three plots in Figure 4.8 tested performance based on the top 5 documents in each collection and the number of terms to be searched. The terms to search for were obtained randomly from the common terms among the collections of the



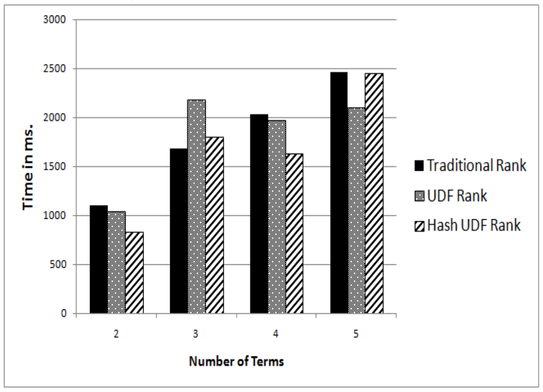
a) Top-5 with 2 Terms.



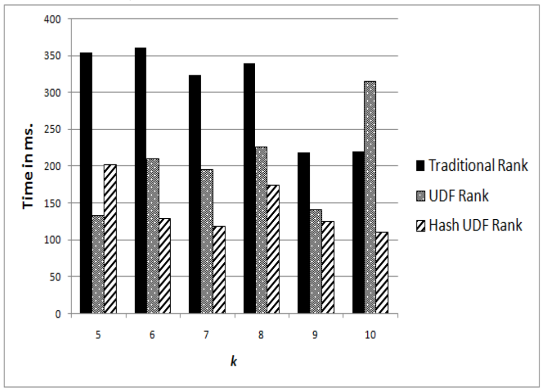
b) Top-5 with 3 Terms.



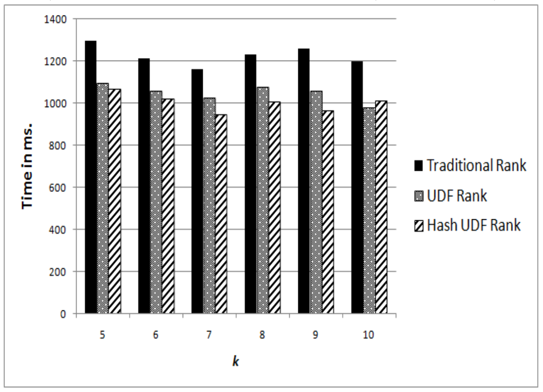
c) Top-5 with 4 Terms.



d) Top-5 varying Terms (ENRON).



e) Top-k varying k, with 2 terms (KOS).



f) Top-k varying k, with 2 terms (ENRON).

Figure 4.8: Top-k Performance.

Table 4.9: Traditional Top-k vs SSQL (time in ms).

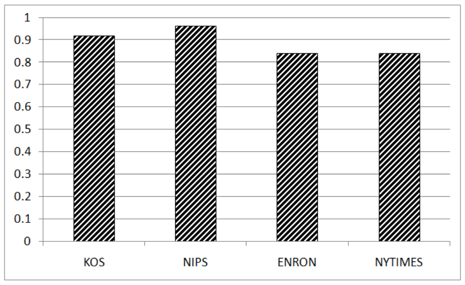
	KOS	NIPS	ENRON
Traditional Rank	212	393	1102
SQL Rank	3237	3160	3160

inverse frequency tables of the collections, and then the average time of execution of five runs was found, discarding the smallest and largest times run. The average of the remaining executions was then computed. The second set of tables in Figure 4.8 compared the algorithms' performances while changing the number of terms to be searched for in a medium-sized collection (ENRON), between 2 and 5 terms. The fourth and fifth figures compared the performance of the three algorithms, changing the number of top-k elements to be searched for in small and large-sized collections (KOS and ENRON). Figures 4.9(a) and 4.9(b) show the accuracy of the final result after pipelining the algorithm in all the collections by keeping constant the three terms to be searched for and the top 5. The NYTIMES collection was selected for analyzing pipelining accuracy, because it is the one that showed the least accuracy of all the collections.

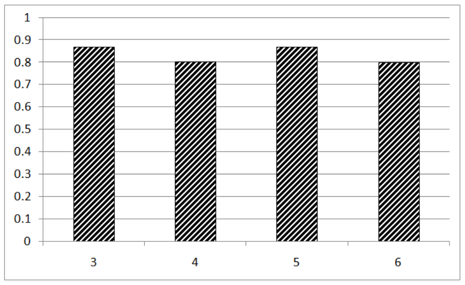
The SQL Rank Implementation performed three to fifteen times slower than the top-k search (see Table 4.9), even though the number of operations was not so different when compared to the other approaches. The rationale behind this is that even though the SQL uses the Rank-Join algorithm and it is indexed by a surrogate key, the number of queries that it sends back and forth between the DBMS and the user application adds overhead to the final execution time. Additionally, the overhead of using ODBC to transfer the queries does not allow this technique to be competitive when compared against the traditional SQL top-k search. In some cases, the SQL implementation performs faster than the traditional query, but the algorithm requires a small top-k and a distribution in the data that allows that the number of joins needed to fulfill the condition be met very quickly. On the other hand, there is a clear advantage because we still cannot visit and evaluate all of the entries in every list, while still working within the hard drive.

We then focused on comparing the UDF implementations and the traditional top-k search. The first three plots in Figure 4.8, (presented in milliseconds) show how the UDF implementations usually run faster than the traditional top-k implementation, independent of the number of terms in the collection size. In particular, the Hash Rank-Join UDF always performs faster in the larger collection, and clearly outperforms the other two algorithms. In Figure 4.8a, the traditional top-k query is the slowest, followed by the UDF implementation, and finally the hash UDF implementation. When the pipelining algorithm takes place, one may how the three top-k queries get much closer in time, because the selectivity in the list decreases. Therefore, the number of documents to manage is much lower, and it appears in some cases that the top-k search can outperform the UDF implementations, when the overhead of the UDF created for loading both tables to main memory and ranking is more costly than a traditional top-k query. The fourth image in Figure 4.8d presents experiments in which the number of terms is changed in the ENRON collection. This figure shows clearly how the number of documents (selectivity) decreases when the number of terms is augmented, and therefore, the top-k implementations tend to perform close to each other. We did not attempt to perform queries with more than five conjunctive terms because the algorithm was not likely to find enough hits with a random combination of five terms in the collections, and the execution of the query containing all of these terms would have ended earlier. This was also because a user search normally involves a small set of terms. Still, the Rank-Join implementation performs faster than the rest of the algorithms, and in the worst case, (5 terms) it performs in the same way as in the traditional search. When we varied the number of top-k terms to be retrieved, we decided to run it in a small data set and a larger data set with different selectivities to observe the performace, using the KOS data set and the ENRON data set, respectively. Figure 4.8e shows how the Hash Rank-Join implementation performs faster on average, than the traditional top-k search, and normally performs faster than the list Rank-Join UDF implementation. Because of the size of the collection and the probability that the selected query would contain just a few documents, there is much variation in the results as compared to a larger collection, as shown in figure 4.8f. Therefore, more stable performance times were observed on larger collection. The Hash Rank-Join implementation clearly outperforms the other approaches. Also, the list Rank-Join outperforms the traditional SQL top-k was increased approach. Once we increased the number of top-k in larger collections, the times remained almost constant. Therefore, we observed that the bottleneck of the problem is the pipelining of the algorithm caused by increasing the number of tables to join, (represented in the first three plots) while we observed a linear growth was observed in the performance as the size of the collection increases.

The last two plots explain how the pipelining approach presented here worked effectively in the collection, comparing the traditional rank results and the UDF



(a) Accuracy in the Collections.



(b) Accuracy varying terms (NYTIMES).

Figure 4.9: Accuracy Measures.

Rank-Join results. Figure 4.9(a) shows the evaluation of the accuracy of the algorithm in each collection. From figure Figure 4.9(a), the results were higher than 80% for all the collections, and higher than 90% for two of them, which we considered to be an acceptable result for this implementation. As discussed previously, this value must be set taking into consideration the distribution of the data. Figure 4.9(b) shows how the algorithm performed in the collection with the least accuracy: NYTIMES. The last figure shows how the final result involving just two lists in the Rank-Join algorithm and retrieving the top 5, has an accuracy higher than 80%, and when the number of terms and join operators increases, the accuracy decreases. Then, when the number of valid join conditions decreases, the accuracy increases again. These results are explained because a higher selectivity in the query (higher number of terms) will result in a much smaller number number of documents. Increasing the percentage of elements that are taken into account in each Rank-Join operation increases the accuracy, but decreases performance. Hence, this tradeoff must be evaluated depending on the distribution of the data in the collection and the accuracy of the top-k results.

4.4.3 Conclusions

The first part of this dissertation is an implementation of an IR engine in the DBMS, mostly in standard SQL. In particular, we takes advantage of recursive queries, SQL optimizations and in-database extensions, such as UDFs and Stored Procedures for document preprocessing, storing, ranking, and retrieving. This approach has not been compared in performance to existing IR-engines since the focus of this work is not to beat the performance of ad-hoc solutions but rather extend the functionalities of a DBMS in order to avoid the cost-prohibitive task of exporting the data outside of a database system. In addition, we are working on some benchmarks with non-restrictive language implementations and RBDMS commercial modules.

This work presented an alternative SQL approach for breaking up the documents without using preloaded tables or DBMS-specific implementations. This algorithm is based entirely on using a recursive query with a tail recursion. Despite the larger overhead compared to using a UDF, the main advantage of such an algorithm is the linear scalability. Continuing with the preprocessing phase, optimizations in the stopword removal step proved to be highly effective, and the computation of a temporary table when we retrieved the relevant documents were for conjunctive or disjunctive queries showed a fast retrieval mechanism for the tested collection. Also, using static and on-demand techniques for inverse document precomputations gives alternatives for amortizing the cost of building the weights in dynamic or static collections. The scalability of the solution proved to be linear, and we consider that these performance results will be similar between different DBMSs, and observed faster results for multi-thread systems. A satisfactory performance was also observed for a medium-sized digital library, and the implementation of the VSM presented here is competitive with the other two models (OPM and DPLM).

In addition to all these experiments, we proved the feasibility of achieving a speed up in the retrieval of the top-k hits with the proposal of a variation of the Rank Join algorithm. We experimented with the Rank-Join approach, and compared the built-in top-k, the newly proposed UDFs' Rank-Join, and SQL Rank Join algorithms for top-k retrieval in the DBMS. The scalability of the three algorithms was found to be almost linear. It was also found that the UDFs variation always perform faster than the Standard SQL and Traditional Built-in SQL implementations. In smaller collections, the traditional SQL implementation performs slower than the Rank Join SQL and the Traditional SQL takes advantage of the query optimizer and performs faster than the UDF Rank Join. Therefore, the UDF Rank-Join improves the current implementation of the built-in (sort-based) algorithm.

4.5 Summary of Contributions

The contributions of the research presented here are unique in the field, since this work is one of few that attempts to perform the entire preprocessing inside the DBMS. This goes along with new schools of thought in the data mining community that exploit the possibility of loading the data first and then performing transformation in an efficient manner. Also, we proved that SQL supports performing all the needed data cleansing tasks and even allows an efficient ranking and complex retrieval of the resulting data.

The result of this chapter is a complete set of data layouts, algorithms, and optimizations for preprocessing unstructured data. These efficient SQL-based algorithms encompass the building of well-known IR models and a new proposal of Top-k Rank-Join-based algorithms for fast retrieval. This is important for the database systems community since the proposed algorithms preserve the separation between the logical and physical level while obtaining acceptable performance. There are, however, areas that still require further research in order to improve the performance of the current algorithms. Part of future work in the area includes exploring blobs for storing inverted indexes, as well as the usage of UDF functions to build IR models. Other research directions include the exploration of extending the built-in operator in a database system to implement one of the Rank-Join variations presented in the current work. This work also opens the door for integrating the preprocessed unstructured data and the structured data already existing inside the relational database system.

Chapter 5

Data Integration

5.1 Integration

The integration of keyword search between a database D and a set of documents C, requires the selection of keywords that appear in both sources. This step requires the creation of data structures that will allow efficient storage and retrieval of the relationships that exist between both sources. In addition it is important that these data structures can be easily analyzed and maintained.

In order to satisfy the flexible querying and efficiency requirements, the process of extracting and querying all of the relationships is the result of an initial relationship construction phase which can also be identified as a preprocessing phase. This preprocessing phase includes the inference and storage of relationships and a posterior analysis of the obtained relationships in order to generate derived relationships.

Finally, these data structures are used during the querying phase.

The preprocessing phase, or relationship construction, obtains all the ρ_γ relationships and derived relationships in an efficient manner. This step is only computed once, and is required for allowing easy updates in the data structure when new documents or rows are integrated in both sources. It is desirable that all the documents never be preprocessed (e.g. removal of stopwords and symbols, among others), and that the algorithm extract all the keyword matches directly by analyzing each document and performing an approximate keyword match. A final step in the relationship construction is the computation of additional data structures used for different ranking techniques (e.g. keywords weights).

The second phase presents different algorithms for querying and ranking the newly discovered relationships. The first two proposed methods rely on the frequency of a relationship given an approximate boolean keyword search in L. The third approach is intended to find relevant keywords related to a document in the corpus. However, this is not a traditional search given that the keywords related to the documents represent the matches between the element in the DBMS and keywords in each document. As a result, the R data structure is used as an inverse index.

5.1.1 Keyword Matching

The relationship inference can be seen as seen as a simple but costly operation defined by querying every element of the database and every document in the collection. However, such an approach is extremely inefficient, and will result in a large I/O

```
Input: $D$, $C$, $\beta$
01 : $L \leftarrow \emptyset$, $E \leftarrow \emptyset$,$R \leftarrow \emptyset$
02 : foreach $k$ in $D$
03 :     if( $k \notin L$ )
04 :         $L \leftarrow L \cup \{< k >\}$
05 : foreach $k$ in $L$
06 :     $t_k \leftarrow 0$,$dt_k \leftarrow 0$
07 :     foreach $d_i$ in $C$
08 :         $A_k \leftarrow \Phi(k, d_i, \beta)$
09 :         if($A_k \neq \emptyset$)
10 :             $R \leftarrow R \cup \{< k, d_i, |A_k|, |A_k|\bar{\delta} >\}$
11 :             $t_k \leftarrow t_k + |A_k|$
12 :             $dt_k \leftarrow dt_k + 1$
13 :     foreach $\rho$ in $R_k$
14 :         $w \leftarrow w/t_k | w \in \rho$
15 :     if( $t_k \neq 0$ )
16 :         foreach $k'$ in $D$
17 :             if( $k = k'$ )
18 :                 $E \leftarrow E \cup \{< k, \gamma >\}$
19 :                 $IEF_k \leftarrow IEF_k + 1$
20 :     else
21 :         $L \leftarrow L - k$
22 :     $IDF_k = \log((N/dt_k) + 1)$
23 : foreach $k$ in $L$
24 :     $IEF_k \leftarrow \log((|E|/IEF_k) + 1)$
25 : return $R, E$
```

Figure 5.1: Relationship Construction Algorithm (Preprocessing).

cost and time complexity. As a result, the purpose of the new algorithm presented in Figure 5.1 is to reduce the number of elements to be searched and maximize the number of relationships that can be found between the two data sources.

The maximization of the number of keywords that can be found in a document was obtained by performing an approximate string matching instead of assuming a perfect match scenario in which a keyword exactly matches a keyword in a document. The

Approximate Boyer-Moore pattern matching algorithm (**ABM**) described in [110, 105] was adapted to search for approximate keywords in the collection of documents in the DBMS. In the rest of this dissertation, the ABM algorithm will be identified as a function Φ. An approximate string matching algorithm requires defining the maximum number of mismatches (edit distance) that a keyword can have in order to be considered a valid approximate keyword. Hence, a β parameter is assigned to guarantee that the number of mistakes is proportional to the size of the keyword instead of fixing the number of mismatches for all the keywords. Moreover, the tuning of the β parameter follows $\lceil |k|\beta \rceil = \delta$, where δ is specified by the user to fit better the elements to match. For example, setting $\beta = 0.1$ allows all of the keywords of size 10 or less to have a maximum of $\delta = 1$.

In Figure 5.1, we present an efficient approach for discovering all the relationships between sources. The input for the algorithm is a Database D, a Corpus C and parameter β for the ABM algorithm. In the first lines, the output sets are initialized to the empty set. R will contain all the matches between a keyword and a document existing in C. L will contain all the different keywords existing between the elements. E contains all the elements of the database and is also initialized to the empty set. The following lines of the algorithm obtain the set of all the possible keywords to look for (without considering γ) and obtain the set of unique keywords L. Despite this, notice that L has still some keywords that are not necessarily in C. The most important loop is the one that iterates through the whole unique set of keywords to search, and finds all the approximate keywords that exist in the corpus of documents. Moreover, additional data structures required for computing the uniqueness of a keyword in both sources can be computed. In line 11, the t_k temporary variable stores the cumulative value of the number of approximate matches that link a document to a keyword. This value is later used for weighting the frequency of the document for every relationship. In lines 16-19, each existing keyword is matched with any existing element in the database. Also, this iteration is used to compute the IEF in a single pass for each k. This step is performed at this stage to guarantee that E occupies the least space possible. Line 21 removes all the keywords that could not be found in the collection of documents. Hence, L will refer only to the matching keywords in both sources. Finally, line 25 returns the only important data structures that are required: a set of all the entities in the database, and the existing relationships and their weights. The last couple of lines (23 to 24) compute the final score for each keyword representing an element in the database. This algorithm creates all of the structures for the three ranking methods discussed here. However, the IDF and IEF structures, as well as δ, can be removed from the algorithm if this value is not used by the ranking method.

Derived relationships are obtained by analyzing the elements found in the database stored in the E data set. It is important to notice that while the matches between the keywords and the documents are contained in R, only E is important for obtaining the derived relationships. Also, a derived relationship does not increase the number of keywords in L. The rationale behind this is that every element in E is already linked to a document. As such, only E is required to extend the relationships. Also, the final derived relationship implies having elements in the database

```
Algorithm 1: Relationship Construction
Input: $E$
01 : $E' \leftarrow \emptyset$
02 : foreach $e$ in $E$
03 :     foreach( $\rho \in R$ )
03 :     if($\gamma = row$)
04 :         if($< k, table_id >\notin E$
04 :           $\wedge < k, table\_id >\notin E'$)
05 :             $E' \leftarrow E' \cup < k, table\_id >$
06 :         if($< k, column_id >\notin E$
06 :           $\wedge < k, column\_id >\notin E'$)
07 :             $E' \leftarrow E' \cup < k, column\_id >$
08 :     if($\gamma = column$)
09 :         if($< k, table_id >\notin E$
09 :           $\wedge < k, table\_id >\notin E'$)
10 :             $E' \leftarrow E' \cup < k, table\_id >$
11 : return $E'$
```

Figure 5.2: Derived Relationship Construction Algorithm.

that do not contain a direct approximate keyword that matches a document. Figure 5.2 describes in detail the algorithm for obtaining these new relationships. The resulting links are stored in a different set E', which is initialized to the empty set from the base relationships. The algorithm for extracting the derived relationships iterates through all the elements in the database linked to the collection. Therefore, in one pass (full scan) in E, it is possible to obtain all the derived relationships by taking every pair e and promoting it to the immediate higher granularity level by the transitive properties of the elements in the database. Verification must be performed to avoid adding an existing relationship in E'.

5.1.2 Keyword Search

Once these relationships have been found, we focus on querying these links stored in R, E, and L. In this paper, we propose three methods for querying the set of ρ and $\rho_{\mathcal{F}}$:

- Method 1: Approximate Boolean Search ranked by γ and frequency.
- Method 2: Approximate Boolean Search ranked by γ and the average edit distance.
- Method 3: Relationship Frequency.

The first two methods are based on querying directly the relationships (approximate boolean search) and raking using γ and frequency. The third method is the result of querying and ranking the documents in C using the inverse document index resulting from the relationships stored in R and a given query Q.

Our first method returns all the ρ ranked by the importance of the element based on γ, the δ between a keyword in L and a $k \in Q$, and finally by the importance of the link between the keyword and the document, where the importance is defined as the frequency of a keyword in each document. For example, let "student" be a $k \in Q$. As a result, the first and second sorting criteria are obtained by analyzing the L structure. Given that EditDistance(k,k'), where $k' \in L$, is valid (less or equal to a given threshold based on β) all the relationships related to k' are returned. As a result, all the relationships in the table name level with a $\delta = 0$ (exact match) are returned first. The last sorting criterion is based on the frequency of approximate match found for a keyword in a document.

The second method also relies on the importance of the element in the database and δ of the keyword to the searched element. However, we use a more complex weighting scheme, in which the importance of a document for a given keyword will be defined as the average of the edit distance multiplied by the frequency of approximate keywords in each document. As a result, a document which contains more frequent matches with a smaller average edit distance will be ranked higher.

Notice in the first two ranking methods, if a set of keywords is given, the result includes any relation that contain any of the keywords in Q. This is due to the fact that we are considering all the keywords to be independent from any association. However, a more complex ranking method can be obtained by querying the documents in the collection given the existing relationships found between the two data sources.

Querying the discovered relationships requires searching only over the newly extracted relationships. Therefore, Figure 5.3 presents an efficient way to traverse the R and E data structures. Method 1 is a reduction of Method 2 by removing $\bar{\delta}$. Hence, we focus on explaining the second querying method. The input of this algorithm is a set of keywords $Q = \{k_1, k_2, \ldots\}$. In the initial lines, the Result set and a partial result container are initialized to the empty set. The first two loops represent a join condition for extracting the valid keyword-document pairs in R that have keywords in a valid approximate distance to a given query Q. Notice that the search can be performed efficiently in L and then joined with the R table. This search can be

```
Input: $Q, R, E$
01 : $Result \leftarrow \emptyset$, $R' \leftarrow \emptyset$
02 : foreach $k$ in $R$
03 :     foreach $k'$ in $Q$
04 :         if($EditDistance(k', k)$)
05 :             $R' \leftarrow R' \cup \{< k, d_i, \delta >\}$
06 : foreach$< k, d_i, \delta >$ in $R'$
07 :     foreach $e$ in $E$
08 :         if($k = e$)
09 :             $Result \leftarrow Result \cup$
09 :                 $\{< \rho, \delta, w_{k,i}\bar{\delta}_{k,i} >\}$
10 : Sort $Result$ by $\gamma$, $\delta$ and $w_{k,i}\bar{\delta}_{k,i}$
11 : return $Result$
```

Figure 5.3: Method 2: Relationship Querying.

performed efficiently by joining using a hash join on *kid*.

All the valid keyword-document pairs are stored in a partial set. Lines 06 to 09 match the relationships with the elements residing in the DBMS. Notice that all of these joining steps are presented in polynomial time. However, this time can easily be trimmed down so that it is almost linear by using hashes or indexes. The conditional statement in line 08 verifies whether there exists a relationship (or derived relationship) between an element in the database and a given keyword. Line 10 sorts the *Result* set by the granularity level, the edit distance of the keyword and the weight of each relationship. The last step returns the sorted set.

The third proposed ranking method is based on the transitive property of the keywords and documents pairs in R. Thus, a rank can be obtained by obtaining a weight for each keyword and applying a ranking model to all the keywords related to each document. In addition, the weight of each keyword is computed to be an indicator of "uniqueness" in the database and the corpus. Hence, this weight of each keyword can be understood as an inverse relation frequency (IRF). The IRF is defined in Equation 5.1 based on the uniqueness of both keywords in the collection, in which $\hat{n}$ is the number of documents that contain k and $\hat{e}$ is the number of elements mapped to k. The first part of the equation is the well-known Inverse Document Frequency (IDF). A second weight is obtained for computing the uniqueness of a keyword in the database. Therefore, the second part of the equation represents the Inverse Element Frequency (IEF). Notice that Equation 5.1 can be summarized as $IRF(k) = IDF(k) * IEF(k)$.

$$irf(k) = \log\left(\frac{n}{\hat{n}} + 1\right) * \log\left(\frac{|E|}{\hat{e}} + 1\right) \tag{5.1}$$

The final ranking of the elements is obtained by applying a ranking function that uses the IRF weight for each keyword. The third ranking method relies on finding how relevant an element is between a database and the collection of documents. As such, a document in the collection can be ranked on how related it is to a given query. In order to do so, the common keywords associated with a document are assigned a weight. Therefore, a document can be ranked using traditional IR techniques. Following this approach, the computed IRF values (which contain the weight of a keyword in both sources) are exploited and a rank is obtained by implementing the cosine similarity formula. In addition to the precomputed IRF values, additional summary tables for computing the normalized table values are required to speed up the computation. For example, a table containing the norm of the document using the IRF values should be precomputed.

Table 5.1: Complexity and I/O Cost of Algorithm 1.

Step	Complexity	I/O Cost
L	$O(m\|T_j\|)$	$m\|T_j\|$
Φ_1	$O(\bar{k} + \beta\bar{k})$	-
Φ_2	$O(\|\bar{d}_i\|\beta\bar{k}(1/(\bar{k} - \beta\bar{k})) + \beta\bar{k})$	-
R	$O(\|L\|\Phi_1 + n\Phi_2)$	$\|L\|n$
E	$O(\|\hat{L}\|m\|T_j\|)$	$\|\hat{L}\|\|T_j\|m$

5.1.3 Complexity and I/O Cost Analysis

A discussion follows of the complexity analysis of the relationship construction algorithm and the first two querying methods. The processing algorithm can be divided into two steps that need to be performed sequentially. The first step is the storage of the unique set of keywords L from the database. The second step is the extraction of all the ρ relationships. A breakdown of the complexity and the I/O cost of all the steps of Figure 5.1 are described in Table 5.1. The first step is the extraction of the unique step of keywords from the database, which is performed in the time it takes to scan all the elements in the database. The second step is the computation of R, which is dependent on the size of $|L|$, the average size of the keyword to search $\bar{k}$, and the average size of a document in the collection. Notice that this step represents the bottleneck of the time complexity of the algorithm. The extraction of all the valid matches between elements in D and E is dependent on the number of keywords with matches in the documents $\hat{L}$ given by ABM. In the worst case $|\hat{L}|$ will be equal to $|L|$. Thus the total complexity is $O(m|T_j| + |L|(\Phi_1 + n\Phi_2) + |\hat{L}|m|T_j|)$. Despite the number of variables in the algorithm, the most important variable is the set of unique keywords to search L.

Table 5.2: Complexity of Algorithm 2.

Step	Complexity	I/O Cost
R'	$O(\|R\|\|Q\|\bar{k}^2)$	$\|R\|$
$Result$	$O(\|R'\|\|E\|)$	$\|R'\|\|E\|$
Sort	$O(\|Result\| \log \|Result\|)$	$\|Result\|$

In Table 5.2, the breakdown of the search algorithm and the I/O costs are described. The first step consists of obtaining the valid relationships that are within a valid edit distance from any of the given queries in the keyword. The search complexity can be optimized by reducing the size of R with a preselection and then performing the loop. The second phase of the search requires matching the elements in the database with the keyword-document pairs. The last step requires a sort based on the edit distance, the granularity level and the importance of d_i. The final complexity of the search is $O(|R||Q|\bar{k}^2 + |R'||E| + |Result| \log |Result|)$, in which early selection and hashes can improve these “joins”. The space complexity of the third ranking method is of based on the number of unique keywords in L since is the maximum space required to store the IDF and IEF computations. The time complexity for computing the IRF is the result of a scan in all the keywords C and id D. Therefore, the time complexity is is $O(|D|+|C|)$. The complexity of the relationship ranking is based on the selected model that exploits the IRF weights.

5.1.4 DBMS Programming and Optimizations

A DBMS implementation requires an efficient storage and retrieval layout. A database schema for storing the relationships is shown in Figure 5.4. The existing elements of the DBMS E are contained in the `table_level`, `column_level` and `row_level`

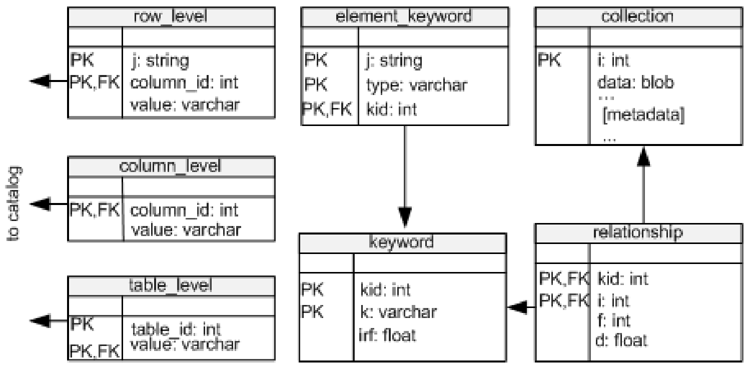


Figure 5.4: Relationship Data Layout.

tables. The format for storing each relationship type has different requirements in order to store their location in the database. As such, a different vertical format is stored for each type. For example, primary keys' values are different among tables. The `element-keyword` table maps all the existing elements e into a keyword. A similar table can be created to store the derived relationships. The L set is stored in the `keyword` table. The document collection is stored in the `collection` table. The resulting keyword-document relations R are stored in the `relationship` table. A final weight is assigned to each relationship based on the number of approximate matches found between a specific keyword and the collection. In order to efficiently extract the type of relationships, indexes must be created in the PK/FK relations. Additional indexes on keyword id kid must be included to speed up the join operations in the retrieval.

The relationship discovery phase requires a process for querying the catalog and finding all the elements that exist in the DBMS. A Stored Procedure that iterates through all the tables extracts the unique set of keywords. An additional process is required to validate whether or not this keyword is a valid element (e.g. numbers are ignored). This step is necessary when traditional search engines discard stopwords or non-representative words in the text. For further reference, see [39]. The relationship discovery process is then performed in two queries in the DBMS. The first query takes the `keyword` table and analyzes each document through a Table Valued Function (TVF). This TVF performs as many of the computations in main memory as possible in order to speed up the execution. Unfortunately, TVF does not allow parallel execution in this context. Hash tables are used as backbone data structures for computing the summarizations required in this step. The following step removes all the keywords without any occurrence. A final query in the preprocessing step is performed to populate the `element-keyword` table and each element table. This query compares the elements in the DBMS to the list with possible matches.

The retrieval of the related relationships for the first two methods is performed with an initial search in the `keyword` table to obtain the valid keywords using a user defined function for the edit distance. The last query obtains the relationships from a hash join based on the keyword id and orders them using the frequency of the number of approximate keywords in each document, including the average edit distance (in the second method). The third method requires the computation of summary tables similar to the ones in traditional ranking techniques to compute the IRF. In order to do so, the IDF and the IEF must be stored in the temporary tables and the final IRF should be stored in the `keyword` table by performing aggregations on R.

5.1.5 Experiments

Our motivation for developing this novel approach was to find a reduced set of relationships in complex database schemes which can be stored, used, and efficiently managed in a DBMS, instead of using the original document collection. As a result, algorithms were developed to extract and query such relationships.

The experimental section is divided into the link extraction and the querying of the extracted relationships tested in two real collections: a water pollution data set and a digital library data set. The first set of experiments will focus on showing how the approximate results vary with the β parameter. Thereafter, the rest of the experiments in the relationship discovery phase are focused on showing the number of elements and links that are obtained in each structure. The second set of experiments focuses on the performance of the three querying techniques.

The experiments were conducted on a computer with an Intel Dual Core processor at 2.53 GHz. The system hardware configuration also has 8 GB of RAM and a hard drive with 1.5 TB of space. The database management systems was a running instance of Microsoft SQL Server on Windows XP. The application was developed using C#, for dynamically generated SQL queries, as a thin front-end for running queries. The relationship construction algorithm and the relationship query algorithm were developed using User Defined Functions (UDFs) and Stored Procedures (SP), as explained previously.

The algorithms were tested using two real data sets in different domains. The first set is a real scientific database containing water pollution data from wells in the

Table 5.3: Texas Water Wells Data Set C and D.

Description	Value	Min	Max
Corpus			
Documents	1000	-	-
Avg. k per d_i	217	1	250
Database			
Number of Tables	32	-	-
Number of Columns	214	-	-
$\|E\|$	36892343	-	-
$\|T_j\|$	111488	0	706223
$\|L\|$	278408	-	-

state of Texas (TWWD), and a collection of public documents downloaded from the Texas Commission on Environmental Quality (see Table 5.3). Note that the database is of a considerable size and complexity (including several PK/FK relations and the corpus is of medium size. The second data set of documents was created from the ACM Digital Library. In addition, a medium-sized database was built from DBLP to associate these data (see Table 5.4). Keywords smaller than three characters, stop words, symbols, and numbers were ignored in both collections. In addition, subsets of these collections $\{100, 250, 500, 1000\}$ were taken to evaluate the performance when varying the number of documents.

5.1.5.1 Relationship Discovery

The first concern when performing approximate keyword searches is related to selecting a good value for β when defining a valid edit distance and the load factor for the algorithm. Tables 5.5 and Table 5.6 shows the effect of modifying the values of β and the ψ factor. When the multisearch algorithm is used with a small β the

Table 5.4: ACM DL Data Set C and D.

Description	Value	Min	Max
Corpus			
Documents	1000	-	-
Avg. k per d_i	206	53	236
Database			
Number of Tables	3	-	-
Number of Columns	9	-	-
$\|E\|$	462778	-	-
$\|T_j\|$	469902	74498	1216779
$\|L\|$	190233	-	-

expected performance of reducing the I/O cost is observed. However, as the value of β increases the overhead of allocating and deallocating several structures and the random access in memory increases the total time when performing such a task. Therefore, this approach shows to be a good candidate for exact matches or small values of β. Table 5.6, it is possible to observe a significant performance increment with values larger than 0.20. As a result, allowing more than a single character for approximate matches reduces significantly the performance of the algorithm. Because we are mostly interested in exact and small variation of these exact matches, this is an acceptable value for our objective.

In Table 5.7, we observe that in these two collections the smallest value of β in which the number of relationships appears to have an improvement in all of the elements in the DBMS is in $\beta = 0.20$. Intuitively, this allows all the keywords with less than five characters to have a maximum edit distance of one and any keyword between 6 and 10 will allow a maximum edit distance of two. It is desirable to have β as small as possible to avoid unrelated matches.

Table 5.5: Times for Varying Load Factor in Φ a 100 Corpus.

	TWWD								ACM DL							
Load	Varying β								Varying β							
	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	15	17	18	18	20	20	23	25	22	31	31	32	33	38	42	48
5	14	19	17	19	22	23	27	32	20	31	31	33	35	45	53	62
10	14	18	18	19	22	23	27	32	19	31	31	33	36	45	55	66

Table 5.6: Times for Varying β.

	TWWD								ACM DL							
Load	Varying β								Varying β							
N=100	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5
	15	17	18	18	20	20	23	25	22	31	31	32	33	38	42	48
N = 250	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	26	35	33	38	37	38	45	51	43	61	65	67	70	84	89	102
N=500	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	39	36	64	72	79	73	94	99	80	106	119	122	129	152	178	188
N=1000	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	98	125	144	145	160	145	190	199	155	216	234	242	255	303	355	376

Table 5.7: Varying β in 1K Corpus.

	TWWD				ACM DL			
β	e in t	e in c	e in r	$\|R\|$	e in t	e in c	e in r	$\|R\|$
0.00	4	25	509738	60817	0	7	36950	61386
0.10	5	25	1623186	7093	0	7	3729	60116
0.15	5	25	1623197	7023	0	7	3752	60512
0.20	5	30	509727	62238	0	7	3376	58517
0.30	5	30	1623127	6971	0	7	3428	57530
0.40	5	30	509626	59140	0	7	3382	54873

The experiments on relationship discovery focus on scalability and performance of extracting the relationships between a database and a corpus. Table 5.8 shows the number of elements in the granularity levels. Notice that only a small subset of elements is required to obtain the elements in a higher granularity level. However, elements in lower levels of granularity require a larger number of documents. In Table 5.9, the amount of relationships between a given collection and a corpus are presented. In contrast to the behavior of the different number of elements, in this case, the number of relationships in each level follows a linear increase with respect to the number of documents. Table 5.10 shows an analysis of the time breakdown of the time performance of relationship discovery algorithm. The first set of experiments shows how the element-keyword relationship does not vary significantly once the size of the collection increases. This is due to the fact that the increase is only produced in the relationship table. As a result, E increases in smaller intervals. Depending on how "descriptive" the schema is, it may happen that a few or none of the relationships will be related to an element in the table name or column name granularity levels. For example, in the ACM DL collection, most of the relationships were found in the row level. In the second set of experiments, the bottleneck of the algorithm for the TWWD and ACM DL occurs in the relationship computation. The rationale behind this is that storing the location of the elements in the database is a costly operation. In addition, we observed that 99% of the time for building the E data structure is spent on obtaining the elements contained in the lowest level of granularity. The algorithm has an acceptable performance when obtaining all of the relationships. Despite this, the performance is highly tied to the size of L, as we

Table 5.8: Relationship Construction.

	TWWD				ACM DL			
Corpus	e in t	e in c	e in r	$\|R\|$	e in t	e in c	e in r	$\|R\|$
	$\beta = 0.0$							
100	5	25	1623126	7394	0	6	35548	6112
250	3	27	604993	15364	0	6	35970	15657
500	4	28	509676	30517	0	7	36400	30190
1000	4	25	509738	60817	0	7	36950	61386
	$\beta = 0.20$							
100	5	25	1623189	6949	0	6	2078	5689
250	3	27	604970	15496	0	6	2456	14222
500	3	28	508273	30436	0	7	2927	27493
1000	5	30	509727	60736	0	7	3428	57530

Table 5.9: Relationships Found ($\beta = 0.0$).

	TWWD			ACM DL		
Corpus	ρ in t	ρ in c	ρ in r	ρ in t	ρ in c	ρ in r
	$\beta = 0.0$					
100	39	331	128921106	0	384	3373533
250	128	814	73951624	0	973	8434298
500	265	1461	100303935	0	1949	16857213
1000	527	3057	197475582	0	3857	33702661
	$\beta = 0.20$					
100	39	285	129332489	0	387	28678
250	128	681	75292424	0	977	68075
500	264	1256	106447827	0	1956	138849
1000	526	2551	209477569	0	3876	296939

observed in the complexity analysis. It is worth mentioning that the preprocessing step is only executed once for every set of documents, and is easily manageable for adding more documents incrementally without having to recompute any previous value.

The experiments in Table 5.11 show the number of relationships derived (to the column level and the table level) using the transitive properties of the elements in the DBMS. This is important because some collections may have ambiguous column names or tables which make them difficult to match. However, the derived relationships are able to find additional links to these elements by analyzing related links. A good example is shown in the ACM DL, in which none of the relationships were found to relate a table name element to any document. Despite this, by analyzing the

Table 5.10: Construction Performance (times in sec).

	TWWD				ACM DL			
Corpus	E	L	R	Total	E	L	R	Total
	$\beta = 0.0$							
100	13	66	430	509	20	5	143	168
250	26	44	418	488	40	5	201	246
500	51	54	650	755	71	6	222	299
1000	101	40	657	798	138	5	242	385
	$\beta = 0.20$							
100	21	45	415	481	35	5	116	156
250	45	36	412	493	75	6	157	238
500	86	43	659	788	138	5	222	365
1000	173	37	659	869	275	5	221	501

final links, a set of new relationships can be exploited to rank these elements. Also, in Table 5.11, one can appreciate that the amount of derivate relationships is only related to the number of unique keywords in both data sets. Hence, adding new documents to the collection will not affect the size of the data structures dramatically, as this is shown in the small increase of entries in the different sized collections.

5.1.5.2 Relationship Querying

In the second section of the experiments, We focus on querying the extracted relationships during the preprocessing step. This is achieved by selecting a set of 20 random queries of author names from the keyword table for each subset of documents and comparing the three methods. Table 5.12 and Table 5.13 show the average, the maximum, the minimum, and the standard deviation performance times required by each set of queries in every collection and subset of documents.

In these sets of experiments, the query times were quite efficient and similar in all the cases for all the ranking techniques. In particular, in the first two methods,

Table 5.11: Derived Relationship Construction.

	$\beta = 0.0$				$\beta = 0.20$			
	TWWD		ACM DL		TWWD		ACM DL	
Corpus	c	t	c	t	c	t	c	t
100	0	23	0	487	0	23	0	422
250	0	24	0	606	0	24	0	521
500	0	25	0	745	0	25	0	659
1000	0	27	0	897	0	27	0	814

Table 5.12: Method 1: Relationship Querying ($\beta = 0.20$, times in ms.).

	TWWD				ACM DL			
Corpus	Mean	StdDev	Max	Min	Mean	StdDev	Max	Min
100	5282	19803	88547	16	23	12	63	16
250	8547	37851	169359	31	28	13	63	16
500	13993	62277	278578	16	35	28	125	16
1000	20741	92439	413469	16	75	80	234	16

the querying times are constant between the set of documents in the same collection. However, as expected, the times increase when the number of elements found increases. The rationale behind this behavior is that the first query in the data set has to verify the keywords that are at a valid edit distance. As a result, a full scan in the `keyword` table is required. Once this step is performed, a join between the relationship table and the `element-keyword` table on the indexed *kid* columns is performed. As a result, the time for the join (hash joined) is quite efficient, and the bottleneck of the query search becomes the early reduction of the keyword table, which requires a full table scan. Notice that the first two methods have very similar performance times and similar standard deviation. As shown in Table 5.14, the third method performs in similar times to the first two methods. Note too, that the standard deviation is much smaller than in the other two methods.

Table 5.13: Method 2: Relationship Querying ($\beta = 0.20$, times in ms.).

	TWWD				ACM DL			
Corpus	Mean	StdDev	Max	Min	Mean	StdDev	Max	Min
100	3602	14313	64188	16	22	9	47	16
250	5589	24594	110078	16	28	12	47	16
500	9626	42698	191031	16	41	41	203	16
1000	20809	92706	414672	31	46	35	125	16

Table 5.14: Method 3: Top-10 Relationship Querying ($\beta = 0.20$, times in ms).

	TWWD				ACM DL			
Corpus	Mean	StdDev	Max	Min	Mean	StdDev	Max	Min
100	148	36	281	109	159	37	266	125
250	155	32	266	125	259	141	719	156
500	169	56	359	125	191	44	297	125
1000	225	79	406	141	207	51	328	125

5.2 Query Recommendation

The query recommendation analysis process presented here is the result of the aggregation of the frequencies of a set of keywords (that have a positive correlation) appearing in a subset of documents. In practice, it is not feasible to obtain the combinations of all the keywords in a set of documents. There are often too many keywords to have a tractable problem. Thus, in order to speed up the execution of the process, we select the $\hat{d}$ most important documents of the collection based on an initial user query. Unfortunately, even with an abridged set of documents, the number of keywords in the documents can still be considered too overwhelming for efficient correlation analysis to be conducted. As a result, the top-k keywords are also obtained from these $\hat{d}$ documents and an importance metric is applied. With these objectives, two different approaches were adopted: using the a-priori algorithm and using an online OLAP algorithm. Since the a-priori algorithm is well known, we will only cover how the OLAP cube algorithm was adapted to efficiently compute recommendations.

5.2.1 Correlation and Metrics

The correlation will represent how related the frequency of one keyword is with another in terms of the whole collection. An efficient one-pass method to compute the correlation values of unique terms in the collection [89]. In addition to the previous definitions, let L be an additional set containing the total sum of all of the different terms in the collection ($L = \sum_{i=1}^{n} x_i$). Moreover, let Q be a lower triangular matrix containing the squared measures between the terms ($Q = \sum_{i=1}^{n} x_i x_i^T$). In the last step, the correlation of a pair of terms a and b is obtained by applying $\rho_{ab} = \frac{nQ_{ab} - L_a L_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}}$.

Once the combinations have been created and verified for having valid correlations, a final analysis of the importance of the concept can be performed by summarizing the occurrences of such combinations of terms in the subset of ranked documents. Three different techniques for ranking these recommendations are proposed here: a single match method, a distance method, and a correlation method.

The **single match** technique is built on performing a single scan of the data counting all the occurrences based on the appearance of the set of terms in a document. Therefore, the occurrence of a combination will be represented by the **MIN** frequency of any of the terms in the combinations within a particular document. A user-defined threshold is used to filter the final results. The **distance match** method, also called proximity match, takes into account the position of the terms in the document and counts the number of times a set of selected terms in the combination appear within a certain user-defined distance, δ. As a result, highly-ranked recommendations will imply that the concept holds a stronger meaning due to the proximity of the terms. The **correlation method** uses the correlation table to make recommendations. In this approach, the validity of a set of keywords depends on its correlation with the initial query. The correlation between each keyword of the user query and each keyword of the candidate set is obtained. From these, an average correlation is then obtained to determine if the keyword set should be recorded. The difference between these three methods is in how the initial query is handled. For the frequency and distance methods, the initial query is used to filter the document collection into a more manageable size. While the same approach is used by the correlation method, it also takes into account the correlation between the initial user query and the keyword set candidates.

5.2.2 OLAP Cube

OLAP techniques can be used to efficiently obtain query recommendations. Figure 5.5 shows a general online algorithm for the computation of aggregations on keywords in the vertical format. In this algorithm, the structure S is a data structure that is used to store the combinations of keywords along with their frequency. The main input table for this algorithm is the keyword table, which always contains a "document id" column and a "keyword" column. The third column, v, is a column that changes depending upon the final filtering method. With such a structure held in main memory, we are able to produce the required combinations and values using only one-pass of the large data set.

```
Input: $level$ , $\hat{t}$
1 : Init data structure $S$
2 : Init $List_{tf}$
3 : foreach<$i, \hat{t}, v$>$r$ in $\hat{tf}$ sorted by $i$
4 :     if $i$ changed
5 :         $List_{tf} \leftarrow \{r.t, r.v\}$
6 :     else
7 :         $C \leftarrow$ GetNextCombo($List_{tf}$, $level$)
8 :         while isCorrelated ($C$)
9 :             if checkTolerance ($v$, $tol$)
10:                 $S_C$++
11:             else
12:                 $S_C \leftarrow 1$
13:                 $C \leftarrow$ GetNextCombo($List_{tf}$, $level$)
14:         end while
15: return $S$
```

Figure 5.5: One-level Online OLAP Cube Algorithm.

The validity of a set of keywords is determined by the filtering method: single match method, distance method, or correlation method. For the single match method, v becomes the frequency of the keyword. For multiple frequencies, the minimum frequency of the keywords was selected to represent the set. The distance method bases the validity of a set of keywords on their positioning. In this case, v represents the position of the keyword in the document. The correlation method bases the validity on the correlation between the initial query and the sets of keywords. The average correlation of the initial query and the candidate set is obtained by retrieving them from the correlation tables. If multiple levels of the lattice are desired, the online OLAP algorithm can take a bottom-up or top-down approach.

Collection	C	Total t	Avg. t per d	$\|tf\|$
ACM	7675	23404	57	0.5M
NYTIMES	299752	92829	226	68M
PUBMED	8200000	134317	58	479M
TWWD	1002	129470	5926	6M

Table 5.15: Collections for Query Recommendation.

5.2.3 Sampling

To further improve the efficiency of the proposed algorithm, the effect of sampling on both performance and accuracy was explored. The first step is to obtain a smaller list of documents, $\hat{d}$, based on $\phi(q, tf)$, where ϕ is the ranking function based on the query and the original set of documents. From this set, we draw τ samples, each having a $\hat{d} = \{d_1, d_2, \ldots, d_m\}$ with replacement from the result of $\phi(q, tf)$. The probability of any document in $\hat{d}$ of being selected in the τ samples is $P(1-(1-1/m)^{\tau})$. Each of the $\hat{d}$ samples of documents is then used as a source for obtaining its top-k keywords $\widehat{t}$ and computing the techniques. Finally, the scores for each combination are obtained as the average of the frequencies. Notice that the set of keywords $\widehat{t}$ is different between samples. Accuracy is calculated as the percentage of total combinations of the top k keywords that are found through the course of the iterations.

5.2.4 Experiments

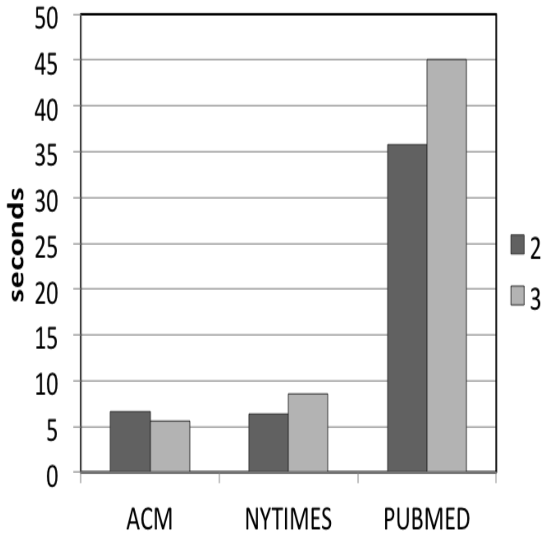
The experiments were performed on an Intel Xeon E3110 server and 4 GB in RAM. The OLAP and the A-priori algorithms were implemented in SQL and UDFs. Four different collections were used during the experiments (see Table 5.15). The first collection is a subset of the ACM Digital Library. The NYTIMES and PUBMED were obtained from the UCI Repository. The Texas Water Well Data Set of Texas (TWWD) was used only for analyzing the quality of the results.

	Single Match		Distance Match		Correlation	
rank	combination	cnt	combination	cnt	combination	corr
1	water, system	368	quality, water	246	information, tceq	0.885
2	tceq, water	368	reduction, requirements	211	information, required	0.848
3	monitoring, water	328	water, surface	170	annual, information	0.848

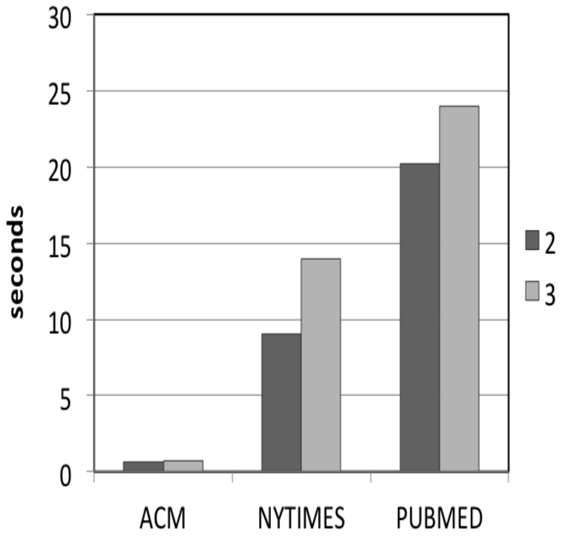
Table 5.16: Recommendations for Query="Information" (ranked by frequency and correlation).

The quality of the results was validated in TWWD with ten initial keyword queries (air, america, dissolved, information, quality, residential, texas, treatment, waste, and water). With these queries, the query recommendations were recorded using the frequency, the distance, and the correlation. Table 5.16 shows the query recommendation returned by the initial query for "information". Of these three approaches, the correlation method definitely returned the most accurate results. In this case, a recommendation is considerated to be accurate if a majority of the returned queries contains data similar to what a user might search for. The recommendations from the correlation method are almost always close to what the user initially queried while those from the other two methods are often more general. For the single and distance methods, the quality of their recommendations heavily depends on the initial query. An average of 25% overlap was found between these two recommendations. The remainder of the queries are very similar to one another, with only subtle differences.

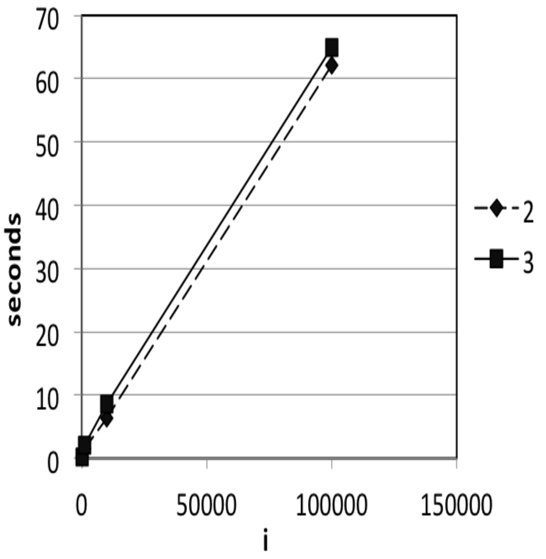
The set of plots in Figure 5.6 present the performance results of the A-priori



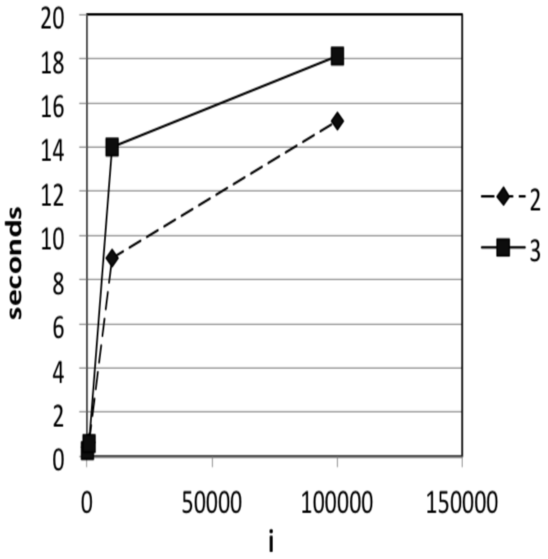
a) Cube: i=10K,topk=5.



b) A-priori: i=10K,topk=5.

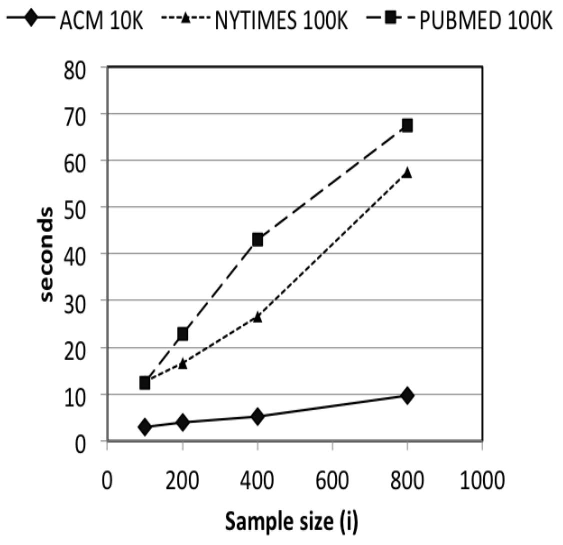


c) Cube: top-5 terms (NYT).

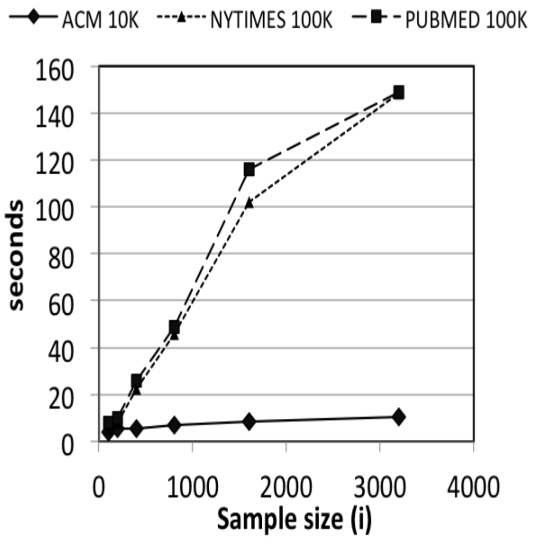


d) A-priori: top-5 terms (NYT).

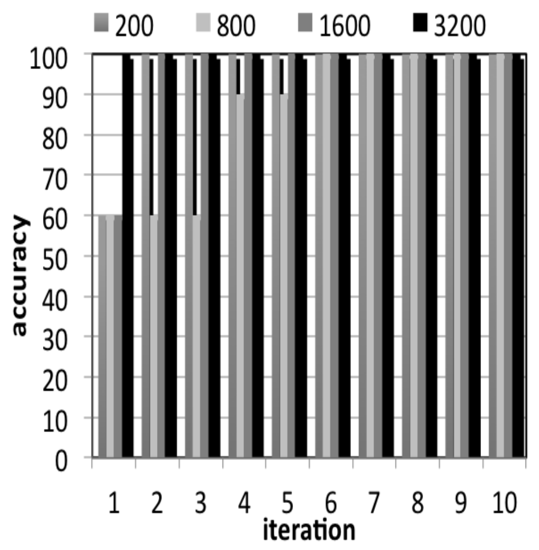
Figure 5.6: Full Document Scan Performance for Levels 2 and 3.



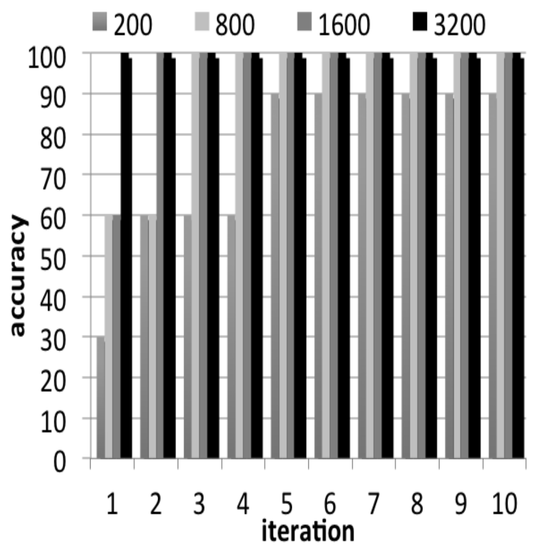
a) Cube: Performance.



b) A-priori: Performance.



c) Cube: Accuracy (ACM).



d) A-priori: Accuracy (ACM).

Figure 5.7: Sampling Performance and Accuracy with Replacement ($\tau = 10$).

and OLAP cube algorithms. Note that the execution time increases with the size of the collection because of the time required to perform the extraction of the $\hat{tf}$ subset. However, because the extraction of this subset is obtained differently in both algorithms, the performance change may also be different. Interestingly, such a difference between the two algorithms reveals some intriguing trends. For example, the OLAP cube algorithm performs faster on collections in which the average number of keywords per document is large. In such situations, the A-priori algorithm is heavily penalized because of the extra matches that the algorithm must perform. Hence, we believe that OLAP proves to be a more stable option due to its ability to be fairly independent of the size of the collection. Having said this, the A-priori algorithm is still quite fast, especially when $\hat{tf}$ can be extracted quickly and managed in main memory. However, the OLAP approach is favored whenever the computations are complex.

Figures 5.6c and 5.6d demostrate the execution time with respect to an increasing the number of documents. The results show that the online OLAP algorithm appears to have a linear reaction. For the A-priori algorithm, when the number documents to be analyzed is small, the performance trend is similar to n^2. However, as the number of documents analyzed increases, the algorithm becomes much more stable, and is significantly faster than the cube approach. For the online OLAP algorithm, performance was expected to be linear (this depends on the number of documents.)

The execution times for 10K documents become unmanageable. Hence, the effect of sampling on both overall performance and safety was analyzed. Figure 5.7a and Figure 5.7b present the performance results for executing $\tau = 10$ iterations. Each iteration would only analyze a small portion of the overall original data set. From these experiments, the A-priori algorithm is slightly faster on data sets with a small average number of keywords per document. In addition, the OLAP Cube algorithm also appears to be more stable than the other options. However, the question arises regarding the validity of using sampling as an approximate solution. Figures 5.7c and 5.7d show results on the number of iterations required to arrive to the final solution. Surprisingly, only a few iterations are needed (normally less than 5) to reach high accuracy. Hence, bootstrapping proves to be a viable alternative for reducing the times.

5.3 Conclusions

This chapter presented a process for discovering "links" between the elements of a central database and a collection of documents without a semantic analysis of the keywords. To the best of our knowledge, this research is one of the few efforts to obtain, manage, and rank links between heterogeneous sources.

These links or references are defined as relationships in the table name, column name and record level of granularity (ρ_γ) that exist against a keyword in a document. In this chapter, a novel and efficient algorithm was introduced during the integration phase that discovers approximate matches between the elements of a database and the unstructured data. The usage of a β parameter during the approximate search was proposed, which provides a fair policy for finding an approximate match based on the length of the keyword. It was found that a small β, such as $\beta = 0.20$, maintains a link focused on exact matches and provides flexibility for small variations of the keyword (since the edit distance in most cases is at most one).

It was also found that that the bottleneck of the algorithm occurs when searching for the elements in the database given a set of keywords. In addition, the parallelization of the search of approximate keywords appears to be a target for future optimizations. Also, the extraction of derivate relationships given the schema hierarchy is presented. Moreover, three methods for searching in the newly discovered relationships proved to be efficient and they each display different perspectives of the extracted links.

The retrieval algorithms for integrated data perform efficiently because the search is executed in small and indexed data structures containing the set of valid relationships and summarized tables. The size of these data structures allow pushing most of the tuples into main memory, resulting in fewer I/Os. In order to reduce the number of I/Os, a data layout, indexing and querying strategies were proposed that optimized retrieval times. Our third retrieval method proposes a document ranking using a novel IRF as a weight for common keywords using traditional ranking to retrieve the top-k documents.

In an effort to improve the retrieval and exploration capabilities of a user in an integrated source, we showed that OLAP can greatly enhance the capability of a traditional search by providing query recommendations. Through this dissertation, it has been shown how the analysis of the intrinsic knowledge of keywords in a collection of documents can be efficiently performed. Specifically, the A-priori and OLAP Cube algorithms were used to provide query recommendations. The experimental results showed that while the A-priori algorithm is slightly faster than the online OLAP algorithm, the latter appears to be more stable in its times. Furthermore, the experiments also show that even with a few iterations, sampling was able to obtain fairly high accuracies. As a result, the experimental results proved that obtaining a few samples with replacement of large collection can achieve a high accuracy. Three main approaches to determining the validity of a set of keywords were created: single, distance, and correlation. The single match method was found to be the fastest of the three methods while the correlation method produces better quality results.

5.4 Summary of Contributions

The proposed algorithms for integrating structured and unstructured data through SQL queries and user-defined functions extend the functionalities of a DBMS, since these algorithms allows performing tasks that are generated outside the DBMS. In addition, the proposed "linking" algorithm discovers relationships that do not need an integration model or rules to work. As such, the matching algorithm is an unsupervised method for integrating data that only requires setting the approximation parameter β. In addition, the complexity of the algorithm is only based on the number of unique keywords to match and the number of documents in the collection.

In a complementary manner, retrieval algorithms for exploring the discovered links are also introduced. The first two algorithms provide a boolean retrieval of the discovered links, which rank the results based on their edit distance to the queried result. A third novel method was proposed that retrieved the results ranked based on the vector space model adapted for exploring links.

Finally, we contributed novel algorithms for query recommendation using OLAP cubes and the A-priori algorithm. This proposal is novel because it explores algorithms that have never been used in this context and provide an enhanced data exploration. The recommendations were obtained by computing one-pass sufficient statistics on a text data set and determining the top results using an OLAP or A-priori like algorithm. The results are the most frequent combinations of keywords that are close to the original query. These results proved that these computations that seem unlikely in database management can be quite efficient by pushing the data structures in main memory, and also can be performed in a sample of the text corpora with similar results.

Future work in this research includes exploring a parallel implementation of the proposed algorithms (only L is needed first), which can speed up the execution of both tasks (preprocessing and retrieval). The optimization for approximate string matching with multiple keywords appears as an interesting area of research, as well. Also, experiments to measure precision and recall of our proposed searching strategies remains as future work. Moreover, the combination of more complex weighting techniques is an open problem for future research. Additional work includes comparing our system to other query recommender systems and trying different policies for finding the occurrences of the set of keywords. Similarly, exploring the effect of injecting knowledge to such query exploration is something that should be explored in more depth. As a result, we decided to experiment with injecting and generating ontologies for OLAP cubes based on heuristics and known taxonomies.

Chapter 6

Knowledge Discovery

This chapter explores preprocessed and integrated sources for extracting non-trivial knowledge from unstructured sources contained inside a database management system. This type of knowledge discovery has always been performed outside of a DBMS, especially, Ontologies and Source Code analysis, which are the focus of this chapter. Moreover, work we developed in the previous chapter is expanded upon in order to extend OLAP algorithms for extracting ontologies from text corpora, as well as for generating OLAP cubes from a given ontology. The final section of this chapter finds references between source code and the schema of a database as a particular case of document integration.

6.1 Extracting Knowledge as Ontologies

This section presents Ontocube, a new algorithm inspired by the query recommendation algorithm presented here, used to extract ontologies from text corpora. Unlike traditional techniques which use some preexisting knowledge or clustering, this efficient algorithm takes advantage of data summarization techniques for finding ontologies by applying a few heuristics based on newly introduced measurements.

6.1.1 Ontocube

While normal applications of OLAP include business reports and financial analysis [19], we believe that the OLAP dimensional lattice can be used to efficiently obtain classes and their relationships. In this case, the data is represented by the collection of keywords and documents, while the level of aggregation allows one to obtain various combinations of these keywords to generate concepts.

Traditional OLAP accepts inputs that are horizontal. However, in the proposed system, vtf has a vertical format, with each row containing a single keyword. This presents a challenge to aggregations because the usual techniques, such as slicing, cannot be used with the vertical format. Despite this, only a small subset is analyzed (the most frequent keywords), $\hat{t}$, of all keywords in the collection. As a result, an algorithm was developed that can obtain the necessary aggregations and perform the required auxiliary computations with the associated keyword frequency (correlations and lift) directly from the vertical format. The correlation will represent how related the frequency of one keyword is with another keywords of the whole collection. In addition to the previous definitions, let L be an additional set containing the total sum of all the different keywords in the collection ($L = \sum_{i=1}^{n} x_i$). Moreover, let Q be a lower triangular matrix containing the squared measures between the keywords $Q = \sum_{i=1}^{n} x_i x_i^T$. In the last step, the correlation of a pair of keywords a and b is obtained by computing equation $\rho_{ab} = \frac{nQ_{ab} - L_a L_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}}$. An additional measure to evaluate the relationship between a pair of keywords is obtained through lift, λ. λ represents how often these two keywords appear in the collection. This value is computed as $\lambda = \frac{n_{a,b}}{Max(n_a, n_b)}$, where n_a and n_b are the number of documents containing keywords a and b, respectively.

6.1.1.1 Ontocube Algorithm

In this section, the ontology extraction process and every optimization performed to obtain classes and build the ontology is explained.

A one-time preprocessing step is required for setting the environment for obtaining the frequency of the keywords. The data set was preprocessed by removing the stopwords and assigning a position for every keyword in every document. The proposed process for extracting concepts is the result of analyzing the most frequent keywords and generating combinations with these keywords. The frequencies of these combinations are also computed, as well as the correlation and lift of these pairs of keywords. The classes are obtained from the combinations by answering these questions:

- Do the keywords appear in the same documents frequently?

- Are the sets of keywords close to each other?
- How often do keywords appear together within the corpus?
- How often is one keyword found in a document but not the other?

The first question addresses the problem of finding keywords that appear in the same documents. In addition to this, one must know if they appear in the same proportion within the collection. As such, the correlation of each combination is computed. The second question, deals with finding how close the keywords appear to each other. This distance between keywords is obtained as the difference between the positions of the keywords. The last two questions attempt to identify those sets of keywords that commonly appear together throughout the collection, or those that do not. The rationale behind this is that the core concepts appear consistently together throughout the collection. On the other hand, there are concepts that may appear strong when present together, but are often found in separate areas. Both of these questions can be answered by observing the lift of the concept.

Efficient summarization is performed in one-pass by creating the keyword combinations "on-the-fly" during the OLAP cube aggregation. The final result of this step is a set of classes from which the ontology will be extracted. The algorithm for obtaining such classes is as follows:

- obtain all available classes by pairing all of keywords within each document,
- for each class, determine the frequency of occurrence and average position gap within the corpus,

```
**Input**: $level$ , $\widehat{t}$
Set of Classes $S$
Init $List_{tf}$
**foreach**$<i, \widehat{t}, p>$ $r$ in $vtf$ sorted by $i$
    **if** $i$ $changed$
        $C \leftarrow$ **GetNextCombo**($List_{tf}$, $level$)
        $DoWhile$ ($C$!=null)
            $S_C$.freq++
            $S_C$.pos+=$r.p$
            $C \leftarrow$ **GetNextCombo**($List_{tf}$, $level$)
        $endwhile$
    **else**
        $List_{tf} \leftarrow \{r.t, r.p\}$
    $endif$
$endforeach$
**return** $S$
```

Figure 6.1: Ontocube.

- filter out all classes whose keywords have a position gap greater than a user-defined threshold, $pDiff$, and
- filter out all classes whose frequency, correlation, or lift does not meet user-defined thresholds.

The pseudocode for the first two steps is shown in Figure 6.1. The remaining two steps can be accomplished by traversing S and removing those classes that do not meet the user-defined thresholds. We are only interested in the classes formed by keywords within a specific document. As a result, it was found to be easiest to first gather all the keywords within a document into $List_{vtf}$ before processing them once the document id, i, had changed. Notice that this algorithm would not generate the combinations of keywords that are not present in the data set.

An efficient one-pass method was adopted to compute the correlation values of unique keywords in the collection similar to the one presented in [19, 37]. The correlation and lift values are computed with the resulting aggregations from the summarization step. As a result, the vtf table is not scanned again to compute both values. Instead, only a summarization table was used. The lift values for each set of keywords are computed from a separate aggregation step wherein the number of documents that each keyword appears in is stored.

The ontology is built from the pool of classes generated from the previous step. Links are then built between the individual classes. Initially, each class is paired with of the other classes within the pool. Let each set of two classes be a relationship. For each of these relationships, the correlation and lift of each set of two keywords was extracted. For example, suppose the relationship $\{A, B\}, \{C, D\}$. It is then needed to obtain the correlation and lift for the following set of keywords: (A,C), (A,D), (B,C), (B,D). These values would show how closely related these two classes are to one another. A high correlation and a high lift from any one of these four subsets would confirm that this is a valid relationship. On the other hand, if these four subsets do not show a high connection, this relationship would be discarded.

Upon completion of this process, a set of valid relationships is obtained, each formed from two classes. The final step is identification of relationships of the form "has a", which define a hierarchy. In other words, we are only concerned with parent-child relationships to build a vertical ontology. The parent is determined with a heuristic based on the frequency of appearance of these classes. We considered a class (A) to be the parent of another class (B) in a relationship if the following criteria is satisfied:

- Number of documents containing A - Number of documents containing B $<$ 10% of total documents,
- Number of classes containing keywords in class A $>$ Number of classes containing keywords in class B.

As a result, if class A passes both criteria, then we consider it to be a parent of class B.

6.1.1.2 Experiments

Our experiments were run on an Intel Xeon E3110 server at 3.00 GHz with a 750 GB in hard drive and 4 GB of RAM. The server was running an instance of SQL SERVER 2005. The OLAP algorithm was implemented entirely with SQL. We tested our approach with a real data set comprised of a corpus of sports articles centered around the Boston Celtics basketball team. There are a total of 50 documents based on game recaps with a total of nearly 15K individual keywords.

Table 6.1 displays the total number of classes that are found when the shown thresholds are applied. These are the classes that form the pool from which the ontologies are later extracted. It is important to note that the total number of possible classes is 786382. Thus, even with minimal thresholds, we are still able to filter more than 99% of the possible classes. It may also be observed that most influential threshold on the number of classes is lift. An important property observed

Table 6.1: Number of Classes Varying corr, lift, and freq.

Corr	*Freq*	*Lift*	*No. t*	*Corr*	*Freq*	*Lift*	*No. t*
0.6	10	0.0	3765	0.8	10	0.0	2383
0.6	10	0.5	146	0.8	10	0.5	48
0.6	20	0.0	1122	0.8	20	0.0	529
0.6	20	0.5	83	0.8	20	0.5	22
0.6	40	0.0	229	0.8	40	0.0	91
0.6	40	0.5	42	0.8	40	0.5	5

Table 6.2: Sample Ontology Result.

Parent	Child	Parent	Child
Boston Celtics	Rajon Rondo	Box Score	Ray Allen
Boston Celtics	Ray Allen	Box Score	Kevin Garnett
Boston Celtics	Doc Rivers	Key Moment	Ray Allen
Boston Celtics	Kevin Garnett	Key Moment	Rajon Rondo
Box Score	Rajon Rondo	Key Score	Kevin Garnett

during the experiments is that the larger the collection of documents, the fewer the number of incorrect classes is obtained. The rationale behind this is that true concepts tend to be more consistent throughout the collection.

Once the appropriate classes have been filtered, the next step in the algorithm is to extract meaningful ontologies from this set of classes. Table 6.2 shows several examples of meaningful ontologies that were extracted. We were able to extract several key players from the Celtics team and also apply the correct parent-child relationship. Rondo, Allen, and Garnett are all players, while Rivers is the coach of the Celtics. The link of Box Score to the actual players makes perfect sense, since each player would have their own set of scores. The last three relationships are the

Table 6.3: Accuracy of Results

Parent	No. Ch.	Acc. (%)	Parent	No. Ch.	Acc. (%)
Boston Celtics	19	89	Key Moment	11	91
Box Line	11	82	Key Score	12	83
Box Score	12	75	Paul Pierce	6	50
Doc Rivers	3	0	Points Rebounds	9	88
Game Celtics	17	100	Rajon Rondo	6	50
Glen Davis	1	100	Ray Allen	4	50
Kevin Garnett	3	33	Score Line	12	83
Key Box	11	0	Score Points	7	43
Key Line	11	0			
Overall	155	67			

most interesting because they provide us with descriptive keywords. From them, one can surmise that the writers of these documents consider both Allen and Rondo to be vital in Key Moments while Garnett is a Key Scorer. Additional verification of these discoveries was conducted and they were found to be true.

It is also important to look at how many erroneous links are formed. To showcase this, the accuracy of our resulting ontologies was examinated. In this case, accuracy represents whether a certain link makes sense or not. For example, Boston Celtics $\rightarrow$ Rajon Rondo is correct because Rondo is a player on the Celtics. Had the relationship been reversed, then this would be considered incorrect. The final tally of the accuracy is shown in Table 6.3. Parents that are players consistently had lower accuracies than parents that represented items or stats. This can be explained by the fact that the documents we obtained were all game recap articles. As such, those articles contain a wide variety of players and teams, but are quite consistent in reporting the scores and stats of each game. The classes Key Box and Key Line obtained 0% accuracy

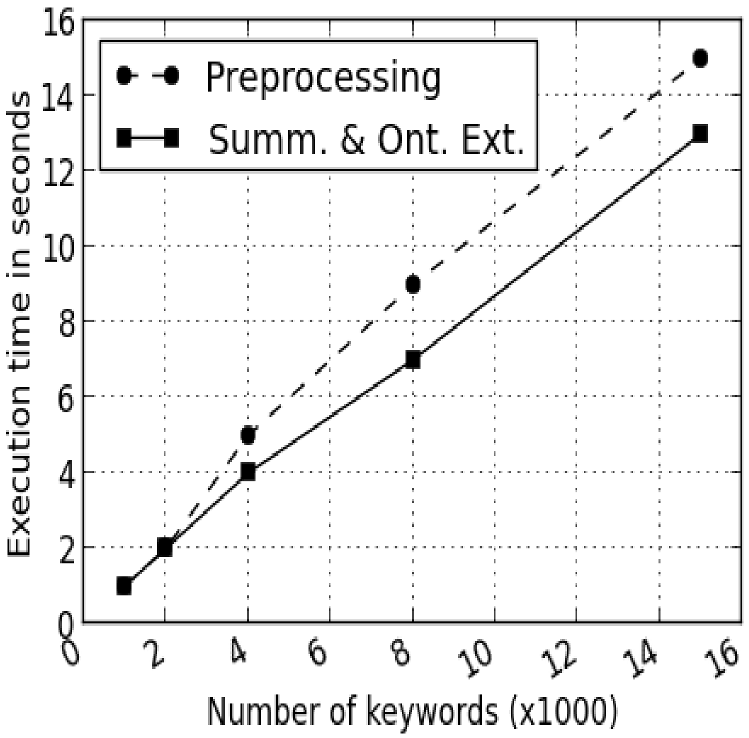


Figure 6.2: Ontology Generation Varying Number of Keywords.

because these classes themselves do not represent viable sets. We believe that with a larger and more varied pool of documents, the accuracies would improve.

Performance is a key factor in determining the usefulness of an algorithm. Thus, two performance indicators are provided. First, Figure 6.2 shows the general trend for both the preprocessing steps and the summarization/ontology extraction steps as the number of initial keywords is varied. One can observe that the trend is linear with respect to the number of keywords. This is to be expected since mostly self-joins and table scans are being used, so the number of keywords should not exponentially

Table 6.4: Breakdown of Computation Time (in secs).

Step	*Time*	*Percentage*
PreProcessing	11.0	40
Corr and Lift	2.3	8
Pairing keywords	11.9	44
Ontology	2.3	8

affect the execution time.

A breakdown of the times for executing our algorithm with an initial keyword size of 15K is shown in Table 6.4. The most heavily-weighted steps are the preprocessing step and the validating pairs step. These two steps were expected to be the most costly because of the self-join that is present in both steps. The preprocessing step includes the NLQ calculation for pairs of keywords, while the other step involves the actual pairing of the keywords to produce the classes.

6.1.2 CUBO

CUBO is an array of data cubes that contain only the existing dimensions within the fact table F given Q. However, the notion of CUBO is based on the premise that a set of documents does not contain all the dimensions in every document. Thus, only a set of "on-demand" computations of the dimensions are required for every document. Moreover, our algorithm takes advantage of this property when computing the aggregation on superior levels in the hierarchy. In other words, CUBO has a lazy policy that waits to perform the computations until it is absolutely necessary. Furthermore, the algorithm avoids redundant computations by focusing only on storing those dimensions that contain attributes to aggregate. If the entire data cube is desired, a post-processing phase can be executed to compute all the missing combinations.

The algorithm for obtaining the CUBO given an ontology $\mathcal{O}$, fact table F, dimensions Q, and a maximum ontology depth T is shown in Figure 6.3. The result is temporarily stored in R that contains a level, combination of classes (combo) and the aggregation measurements. Furthermore, the entire computation of all of the data cubes is computed through a single scan over the filtered data set. CUBO is composed of three major steps: load ontology, computation of on-demand combinations and the data cube aggregation. The final step of our algorithm stores the resulting data cubes into a relational table. **Example 2:** Given an ontology with the following characteristics $h = 3$, $k_3 = 2$, $k_2 = 1$, and given that all the dimensions share parent $H_{(1,1)}$, the resulting CUBO is $\{\{\{D_1, D_2\}, \{D_1\}, \{D_2\}\}, \{\{H_{(1,1)}\}\}, \{\{H_0\}\}\}$.

6.1.2.1 Ontology Loading

Loading the ontology in main memory is the initial step for the algorithm presented here. The format of the ontology is language-specific (OWL is used in this case). However, this algorithm builds a balanced internal tree-like representation of the given ontology stored as a set of linked lists (every node has a parent pointer). This in-memory loading is important for avoiding joins with dimension tables and allows a fast hierarchy traversal. Due to the fact that we are only interested in capturing class and relationships, all of the properties and types of relationships are ignored

```
   Input: $\mathcal{O}$,$F$,$Q$,$T$,$\{A_1,...\}$
   Output: R
1  R ← $\emptyset$;
2  $\mathcal{O}$ ← LoadOntologyFromOWL();
3  $\hat{F} \leftarrow \{t_i | t_i \in F \wedge \exists D_j \ s.t. D_j \in \ t_i \wedge D_j \in Q\}$ ;
4  t ← $\emptyset$;
5  while row in $\hat{F}$ do
6      if document changed then
7          R ← R ∪ BuildCube(t,$\mathcal{O}$,T,R,$\{A_1,...\}$);
8      end
9      t ← t ∪ $\{D_j\}$; /* $D_j \in row$.                         */
10 end
11 BuildCube(t,$\mathcal{O}$,T,R,$\{A_1,...\}$);/* Adds last doc.       */
```

Figure 6.3: CUBO

during this process. Ontologies that are equivalent to flat or balanced hierarchies are easy to capture and traverse. By counting the number of parents, it is possible to know the level. However, an unbalanced ontology must be captured in a tree representation that preserves the level for each node in the tree. Otherwise, the algorithm will return incorrect results by pairing dimensions that are not within the same level. In order to avoid this problem, dummy nodes must be included in the tree data structure, and must be ignored during the combination computation and aggregation. For example, if a dimension D_1 does not have a parent in the immediate superior level $h-2$, in which dimension D_2 has parent $H_{(h-2,1)}$, a dummy parent node must be added for D_1 in order to allow the algorithm to traverse the tree all the way to the root. As a result, if the user desires to support unbalanced ontologies, the given ontology should have a way to provide a property that specifies the level of a dimension in the tree. In this algorithm, hierarchies with multiple parents are not covered, but it is possible to extend the algorithm to handle such cases. Extensions to the algorithm also include adding a new indirection level in the $h-1$ level of the R structure to manage the summarization of instances.

6.1.2.2 On-demand Combinations

The lazy-policy of only working with those combinations that are present in the data set is critical for efficient performance in a space data set. Once the ontology has been loaded into main memory, the fact table F is filtered to only focus on those dimensions that are in Q. A single scan through the data set is performed to read all the entries containing a document t_i and a dimension D_j. In order to process a document t_i, all the dimensions must be collected before proceeding. With all the dimensions of a document collected, the algorithm computes the combinations possible with this set of dimensions and stores them in a list for accumulating all of the desired measurements. The incoming attributes are always sorted in order to guarantee that the combination will be unique when storing the resulting combination in R. The sorting, as well as the combination computations, are relatively inexpensive operations due to the limited amount of dimensions to consider per document. However, if the data set is dense, these tasks will represent the bottleneck of the algorithm.

6.1.2.3 Data Cube Aggregation

The data cube aggregation will cover storing every combination in the corresponding data cube (level) and traversing the loaded hierarchy all the way to the root H_0.

This function is covered in the `BuildCube` function shown in Figure 6.4.

```
    Input: $t$,$\mathcal{O}$,$T$,$R$,$\{A_1, \ldots\}$
    Output: R
 1  $s_h \leftarrow$ Combos();
    /* Aggregate all the existing combos of the h-1 level.          */
 2  foreach combo do
 3      R $\leftarrow$ R $\cup$ $\{1, combo, \{A_1, \ldots\}\}$;
 4  end
    /* Recursive function to extract all unique concepts by level h-2
       to 0.                                                         */
 5  $s_{0,\ldots,h-2} \leftarrow$ CombosForOntologyLevel(s,$\mathcal{O}$,T);
    /* Increments found combos by level.                             */
 6  foreach $l$ in $s_{0,\ldots,h-2}$ do
 7      foreach combo in $s_l$ do
 8          R $\leftarrow$ R $\cup$ $\{l, combo, \{A_1, \ldots\}\}$;
 9      end
10  end
11  return R;
```

Figure 6.4: BuildCube

The `BuildCube` algorithm takes the result of the on-demand combinations of each document and stores the result in main memory. Then, it traverses the ontology tree for every dimension D_j in t_i. The corresponding combinations are accumulated in R, in which if a corresponding combo does not exist, a new entry will be created. Otherwise, the corresponding measurements are aggregated. For each D_j that exists in each document, all of the superior H are extracted, and their combinations are computed and stored in $s_{0,\ldots,h-2}$. Finally, the corresponding value is accumulated for each existing combination in every level. The `CombosForOntologyLevel` function is a recursive function that will stop after a certain number of levels T has been reached, or the parent is null. Due to the fact that balanced trees are always considered, it is possible to perform this operation in a simple manner. Support for dummy nodes should be considered in this section of the algorithm. Once the resulting data cube R is computed, R can be stored as relational tables inside a database management system.

6.1.2.4 Complexity

The complexity of the algorithm is given by the time it takes to compute all the combinations of the dimensions for each document $n2^k$. Moreover, because the ontology representation is always the result of balanced hierarchies (remember the introduction of dummy nodes), the time it takes to traverse an entire branch of the tree for each dimension requires h steps. Thus, the total complexity of the algorithm is $O(n2^{kh})$. Despite the time complexity of the algorithm being larger than the traditional data cube computation that has a complexity of $hn2^k$ (because there is a need to compute a data cube for every level), in practice, the average size of k per document is small (around 1 or 2) and h is almost always small (less than 5). This results in a faster performance because there is no need to concentrate on computing dimensions that are not within the data set.

The space complexity of the algorithm is limited only by the number of dimensions per level in h. If the data set is dense, the space required by the algorithm is given by $\sum_{l=0}^{h} 2^{k_l}$, where k_l is the number of dimensions per level. Therefore, in the worst case scenario, the space complexity of the algorithm is of the order $O(2^k)$.

6.1.2.5 Integration with a DBMS

The fact table F and the filtered table are assumed to be cluster indexed by i. This is an important assumption for guaranteeing that all the dimensions of a document are contiguous in one block. Extending a relational database management system to support the algorithm requires injecting the algorithm as a routine programmed in a procedural language (e.g. C or C#). In order to do so, database extensibility mechanisms (e.g stored procedures or table-valued functions) can be used to achieve this goal. The main advantage of using an extensibility mechanism is that it is possible to maintain the ontology, as well as the CUBO structure, in main memory. Database extensibility mechanisms provide a framework to hold data structures in main memory while scanning a data set in a cursor fashion. Unlike other types of user-defined functions, such as user-defined aggregates (UDAs), a stored procedure or table-valued function that requires processing every row will not offer the parallelism that is native to UDAs by default. Figure 6.5 shows a stored procedure call to execute the CUBO algorithm in a DBMS.

```
EXEC CUBO 'D:\ontology.owl', 'dataset','D1,D2,D3'
```

Figure 6.5: Stored Procedure SQL Call.

The extensibility mechanism contains a SELECT statement that filters the fact table F using a WHERE/IN clause to consider only those dimensions in Q. This will be the only pass through the data set. Notice that this query is represented as two steps in the algorithm. The backbone data structure R that contains all of the aggregated attributes $A_1, \ldots, A_e$ for all the levels in the hierarchy is an array list of hash tables, wherein each hash table contains a data cube.

Unfortunately, there are limitations associated with every relational database management system (which limits the portability of a stored procedure or user-defined function implementation). For example, the amount of data that can be maintained in main memory at one time, the ability to access and read external files (used to load the ontology), or the possibility of creating and storing the resulting data in a relational table are some limitations.

6.1.2.6 Experiments

To verify that the algorithm presented here performs better than a traditional approach, as well as to scale to large datasets and dimensions, the algorithm was tested on two databases. The experiments were run on an Intel Xeon E3110 server at 3.00 GHz with a 750 GB in hard drive and 4 GB of RAM. The server was running an instance of SQL SERVER 2005. The application was developed entirely as a stored procedure in C#, as part of an extensibility mechanism from SQL SERVER 2005.

The databases used for testing the application included an ontology in OWL format and a real and a synthetic data set stored in a relational table F. The synthetic data set is a materialized view from the TPCH data set. The view is the result of $Lineitem \bowtie Part$, wherein each row represents a dimension in a document (in fact, a document can be seen as a transaction). The real data set is the dbpedia project, which contains 3 million abstracts with 308 dimensions extracted from the Wikipedia project. Every D_j in this project represents a classification topic for each document

Table 6.5: TPCH Corpora.

n	Avg k	Max k	Min k	Total k
1K	1	3	1	1038
10K	1	3	1	6589
100K	1	5	1	9702
1M	1	5	1	9702
10M	1	5	1	9702

in dbpedia. Furthermore, A_1 was assigned to be one for all the D_j as our purpose is to count the number of documents per topic. This data set was preprocessed and stored in a relational table F. The preprocessing phase required almost two days to extract the classes existing in the original abstracts. Both databases were given an OWL ontology of 2MB and 1MB in size, respectively. Both ontologies had a depth of 3 levels. A full description of the databases is given in Table 6.5 and Table 6.6. All of the performance experiments were repeated five times and the results were averaged after removing the upper and lower quantities. The dimensions in Q were obtained randomly for every run from the pool of all possible classes in the $h - 1$ level of the ontology. In addition, the default databases for testing the algorithms were the original 3M dbpedia data set and the 10M TPCH data set. All of the times are in seconds unless otherwise specified.

Initially, we focused on studying the performance of our algorithm as compared to the traditional cube implementation from SQL SERVER 2005. In order to perform this experiment, the vertical dbpedia data set F had to be modified to have the classes as column names and the documents as rows. Hence, the experiments were set to use this horizontal layout for the traditional cube operator and the vertical

Table 6.6: dbpedia Corpora.

n	Avg k	Max k	Min k	Total k
1K	2	9	1	156
10K	2	14	1	231
100K	2	16	1	263
1M	2	26	1	302
10M	2	46	1	308

Table 6.7: Performance of Traditional Cube and CUBO (* unable to compute)

d	Traditional Single Level	CUBO
2	36	5
4	36	8
8	37	9
16	*	15
32	*	44
64	*	96

layout was used for CUBO. In addition, the traditional cube was only performed in the lowest hierarchical level due to the lack of native support for hierarchies. Obtaining a similar result with the cube (or SQL queries) would require loading the ontology in a star schema that should be known upfront, and then running the cube operator several times for every level. The results, presented in Table 6.7, show that our algorithm performs an order of magnitude better than the traditional data cube operator in only the lowest level of the hierarchy (level $h-1$). Moreover, the traditional data cube operator cannot scale to a larger number of dimensions to compute. This is due to the fact that the traditional data cube is also computing those dimensions that have zeroes. This experiment was not performed in the TPCH data set because this data set exceeds the maximum number of dimensions allowed

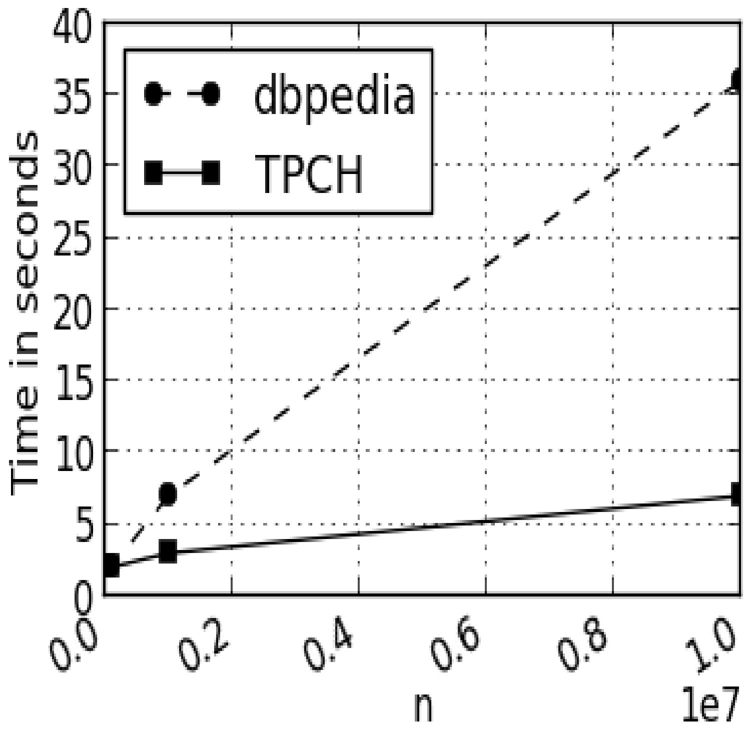


Figure 6.6: Varying Corpus Size.

in a relational table.

Figure 6.6 presents scalability experiments by varying the size of the corpus in the synthetic and real databases. The size of Q was fixed to ten, and the corpora included a range of collections from 1K to 10M of documents. The experiments showed that CUBO scales linearly based on the number of documents in the collection. However, the speed of the increase is related to the average number of dimensions per document. Therefore, a data set with a smaller average number of dimensions per document will be processed faster than one with a larger average of dimensions.

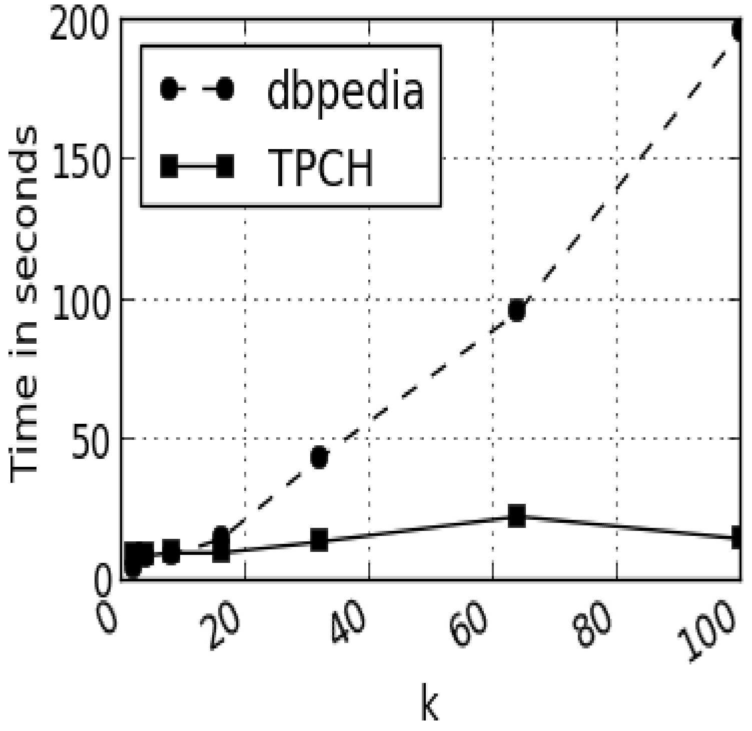


Figure 6.7: Varying Number of Dimensions.

Additional scalability experiments for the algorithm include modifying the number of dimensions in Q. Figure 6.7 shows the results for the TPCH and dbpedia databases. The results perform as expected in the complexity analysis discussed in subsection 6.1.2.4. The plot shows an exponential increase related to the average number of dimensions per document. Hence, dbpedia has a larger exponential growth than the TPCH data set. This is due to the larger average number of sparse dimensions per document in dbpedia than TPCH. A lower number of average dimensions per document results in the avoidance of computing large combinations and a decrease in the exponential growth (shown in the TPCH data set). The inflection point of the function is the result of the selectivity of dimensions in the corpora;

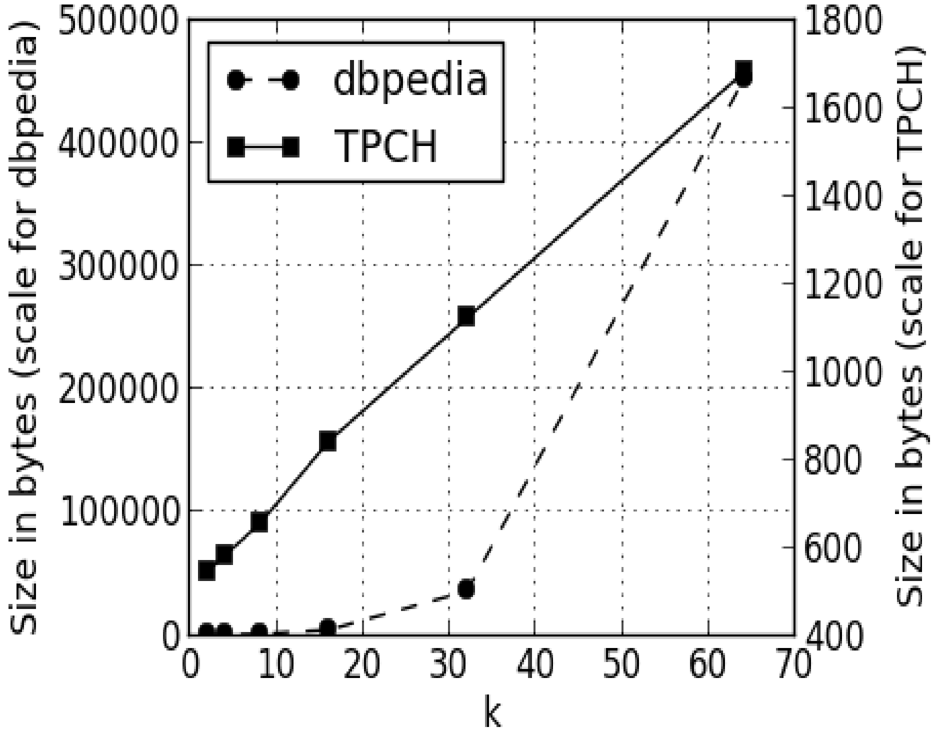


Figure 6.8: CUBO Size when Varying Number of Dimensions.

hence, this point depends on each data set.

Experiments showing the space complexity of the algorithm were also run. Figure 6.8 presents the results of this experiment for both collections. Notice that the axes have different scales due to the sparsity of the databases (TPCH is more sparse). The dbpedia data set shows a clear exponential growth of the data structure size. In a similar manner, TPCH has a similar growth that is almost undetectable due to the sparsity of the data set. It was also found that the maximum k that could be computed was around 100 dimensions and an ontology input file of around 2 MB. However, this limitation is DBMS-specific.

Table 6.8: Varying Ontology Levels in dbpedia (time in seconds).

n	ALL	MAX 2	MAX 1
1K	2	2	1
10K	2	3	1
100K	2	3	1
1M	7	6	5
10M	36	28	25

Table 6.9: Varying Ontology Levels in TPCH (time in seconds).

n	ALL	MAX 2	MAX 1
1K	2	2	2
10K	2	2	2
100K	2	2	2
1M	3	2	2
10M	7	7	7

Further experimental analysis of complexity of the algorithm is shown in Tables 6.8 and 6.9. These tables show the performance results of modifying the number of levels that are taken into consideration from both ontologies and databases. The results showed that in both databases, the overall impact of modifying the level in the hierarchy is minimal when the average k is small in all of the text collections. However, there is an almost linear increase for the real data set in the largest of the collections. It is then possible to observe that the experimental results support the theoretical upper bound of $O(n2^{kh})$, in which there is an exponential growth. Due to the lazy policy of only computing those dimensions that are present, in practice, the complexity of the algorithm is followed by an almost constant performance (e.g. there are no combinations involving all the k dimensions).

Table 6.10: Profiling dbpedia (time in seconds).

Task	Time	Percentage
Load Ontology	0.017	0%
On-demand Combinations	0.030	1%
Data Cube Aggregation	0.061	1%
I/O	5.891	98%
Total	6.000	100%

Table 6.11: Profiling TPCH (time in seconds).

Task	Time	Percentage
Load Ontology	0.068	1%
On-demand Combinations	0.001	0%
Data Cube Aggregation	0.010	0%
I/O	6.921	99%
Total	7.000	100%

Finally, a profiling analysis of the algorithm in both databases is shown in Tables 6.10 and 6.11. The experiments are consistent regardless of the data set and show that obtaining the combinations is the least costly step, along with storing the results in the hash table. However, reading and building the ontology in main memory takes longer than building the actual CUBO. As expected, the majority of the processing is spent in access to secondary storage.

6.2 Integration and Querying of Programs and Schemas

QDPC is a system that allows joint keyword searches and complex analysis of the resulting matches to be performed. As a result, QDPC is composed of an integration phase and an extraction phase. The system presented here exploits a relational database as a backbone for efficient extraction and retrieval of matches. In other words, the entire process of managing both sources is performed within the DBMS. Therefore, both modules rely on optimizations that exploit either SQL or UDFs, depending on the required task. In Figure 6.9, we exemplify the relationships between all the elements of S and P. Table 6.12 shows in detail several matches present in Figure 6.9.

Table 6.12: Example of Matches between S and P in the Context of Water Quality.

name in S	parent in S	type in S	keyword in P	distance	type in P	loc. in file	file
Pollutant	null	table_name	Pollutant	0	metadata	filename	Pollutant.java
Pollutant	null	table_name	Pollutant	0	SQL	SELECT * FROM Pollutant	Pollutant.java
ptype	Pollutant	column_name	ntype	1	var	String ntype = ””	Pollutant.java

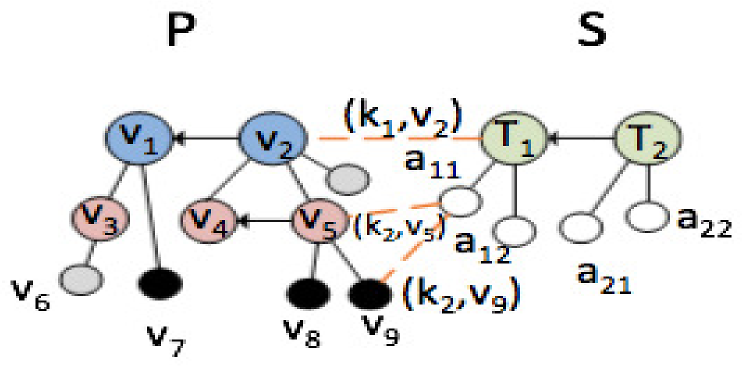


Figure 6.9: Matches between a Program P and a Schema S.

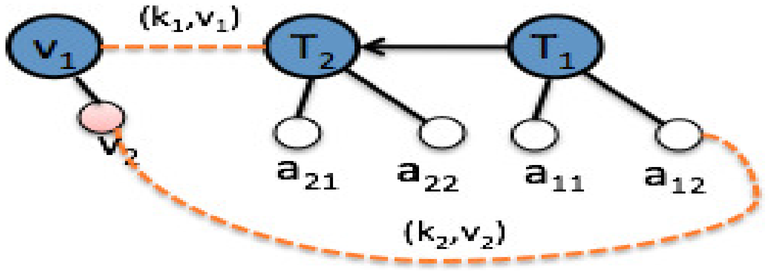


Figure 6.10: Resulting Graph.

QDPC also takes a query Q which is composed of a set of keywords $Q = \{k_1, k_2, \ldots\}$ in order to perform a uniform search in both sources. The searches are performed over a set of approximate matches between P and S mapped to a keyword. An approximate match of a keyword k is obtained based on an edit distance function edit(k,k'), where the edit distance represents the minimum number of insertions, deletions, or substitutions required to turn k into k'. Therefore, we consider an approximate match if and only if $\texttt{edit}(k, k') \leq \lceil \beta * |k| \rceil$, where β is a real number in the interval [0,1]. Intuitively, the value for β cannot be larger than 0.5 because it will accept many unrelated keywords as valid approximations. In the particular case in which exact matches are the only ones desired, the value of β equals zero. The value of β should be tuned based on the programming style in the source code problem. Because exact matches are clearly the more important ones, a small value of β is normally desired (e.g. between 0.10 to 0.20). A match is represented as a pair of the form (k, v). Finally, the result of a query Q is all the graphs, G, containing all elements (e.g. tables, columns, class, methods, variables, SQL queries) that are required to satisfy all the approximate matches between P and S.

For example, as shown in Figure 6.10, let S_1 have two tables $T_1(a_{11}, a_{12}, a_{13})$ and

$T_2(a_{21}, a_{22})$, where there $T_1(a_{12}) \rightarrow T_2(a_{21})$; and P_1 has a single class:

Given that $T_2 \mapsto k_1$ and $a_{12} \mapsto k_2$, and $\texttt{edit}(k_1, v_1) \leq \lceil \beta * |k_1| \rceil$ and $\texttt{edit}(k_2, v_2) \leq \lceil \beta * |k_2| \rceil$, then (k_1, v_1) and (k_2, v_2) represent a match between S_1 and P_1. If a query is given involving keywords k_1 and k_2, then the resulting graph is $G_1 = \{T_2, T_1, a_{12}, v_1, v_2\}$.

```
public class v1
{
      int v2;
      int v3;
}
```

6.2.0.7 Integration

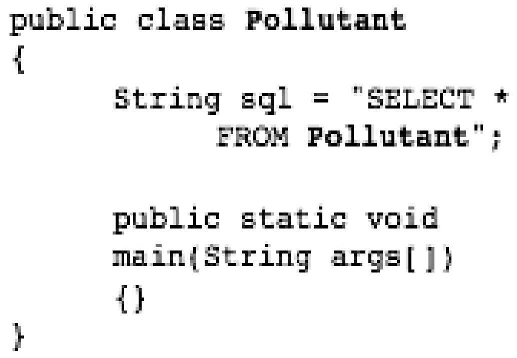
The integration is a refinement of an earlier idea presented in [40] for managing structured and unstructured sources. However, the analysis was limited to simple queries. The integration phase can be summarized as:

- Preprocessing
    1. Analyze source code files to extract all v.
    2. Extract control dependencies.
- Integration.
    1. Obtain unique keywords.

2. Search for approximate matches in the preprocessed source code.
3. Apply indexes to the resulting tables.

The notion of achieving the integration relies on a two-step process. The first step preprocesses all the source code files to extract all the metadata, class names, method names, and variables, as well as the calls that exist between them. This will generate three tables: a source code table (*P_document*) which contains an identifier for each $v \in P$ and the location in the file, *P_mcall* (caller method, method) which has the control dependencies between methods, and *P_ccall* (caller method, method) which has the control dependencies between classes. Notice that the P prefix indicates that a table contains information regarding a program P. The integration step will rely on three main tables *M_predicate* (keyword id, keyword), *S_database* (keyword, location, parent) and *M_document* (keyword id, edit distance, type, parent method, parent class, location in file), in which *M_document* will hold all the approximate matches to the source code files. Similar to the P prefix, the S and M prefixes relate to the schema and matching results, respectively.

The first task during the integration will focus on obtaining the unique keywords of the database's schema and finding all the approximate matches that exist between these unique keywords and every v in P. Therefore, this process is bounded by the number of unique keywords and the number of files. During this integration phase, a batch of keywords is tested one a time. As a result, if the number of unique keywords in S is small, the algorithm is only bounded by the number of discovered elements v in the source code repository P. In order to find approximate matches, the Approximate Boyer-Moore algorithm (ABM) was used [110]. The rationale behind



```
public class Pollutant
{
      String sql = "SELECT *
            FROM Pollutant";

      public static void
      main(String args[])
      {}
}
```

Figure 6.11: Source Code Matching.

this is that by using this algorithm, it is possible to capture valid text patterns within the desired edit distance. For example, in Figure 6.11, the string "pollution" is found as a class name and as a SQL query.

Once this discovery phase has been finished, indexes are applied to the summary tables containing the approximate matches. Indexes were also added to the unique keywords, source code, and metadata summary tables.

Matches between the database and the source code files are discovered as follows: a set of all the elements from the database (T and a) is queried from the catalog in order to extract all of the unique keywords. This set of keywords will be stored in two summary tables. *M_predicate* will contain all of the unique keywords (obtained with a UNION query) and *S_database*, which will contain the location of the elements in the database. The keywords in *M_predicate* are then matched between every element in the database and then for every keyword k in the code repository P. The resulting matches in the source code are stored in a *M_document* table. Table *M_predicate* is pruned to only hold those keywords that are represented in the matches. This matching is performed through database extensibility features, such as using user-defined functions (UDFs), in order to keep in-main memory all of the data structures used to obtain an approximate distance (using edit distance) for every concept and every given string coming from the database or collection of source code files. This matching step is the bottleneck of the keyword matching. However, to ease the processing, the matching is performed in batches. In other words, a set of keywords k mapped from S is scanned simultaneously to avoid multiple passes over the *P_document* table and reduce I/Os.

Tables *S_database*, *M_document*, and *M_predicate* (summary tables) are created to avoid repeated searches on a particular data source and they include the location of the concept in the database and in the source repository. By using these summary tables, it is possible to obtain the matches (stored as a view called *M_table*) by joining these three tables, in which there is a hash join on the matching keywords between *M_predicate* and *S_database*, and a hash join on the id of v present in *M_predicate* and *S_database*.

This query is quite efficient due to the keyword indexes. An example of the resulting matches is shown in Table 6.12. Furthermore, the data type is considered irrelevant when matching metadata, but this remains something to be explored in the future.

The complexity of the integration is given by the number of unique keywords to search and the number of files in P. Therefore, the integration is on the order of $O(rn)$, where r is the number of unique keywords and n is the number of files.

6.2.0.8 Querying

The exploration is conducted only by analyzing the summarization tables obtained in the previous phase (mainly the approximate matches in the *M_table* view).

Analytical queries are obtained mostly from the predicate table and the approximate match table as shown in Equation 6.1. These analytical queries are efficient one-pass aggregations in SQL using functions such as SUM, COUNT, MAX, and MIN.

$$\pi_F(\sigma_Q(M\_predicate) \bowtie (M\_document)) \tag{6.1}$$

These queries are able to answer questions dealing with the number of matches that are associated with a particular v, T or a, as well as which matches are present if the aggregation (π_F) is removed.

Furthermore, these aggregations may be taken one step further and OLAP cubes may be computed from these matches by generating aggregations with different dimensions that include all the granularity levels inside the DBMS and within the source code. An OLAP query that exploits the summarization tables based on the CUBE operator from SQL SERVER is shown in Figure 6.12.

This query can answer complex analytical queries such as "Which column a has the highest number of dependencies associated with v?"

Finally, the most complex searches are those that require following the dependencies. These searches answer queries such "What are all the methods associated

```
SELECT
     CASE
          WHEN GROUPING(a) = 1
          THEN 'ALL'
          ELSE a
     END a,
     CASE
          WHEN GROUPING(v) = 1
          THEN 'ALL'
          ELSE v
     END v,
     SUM(f) AS f
FROM M_TABLE GROUP BY a, v
WITH CUBE;
```

Figure 6.12: OLAP Cube Query.

with a particular column?" In order to find these dependency graphs G, we propose the following algorithm: The computation of the graph in S is performed efficiently in main memory due to the reduced number of elements. However, computing the resulting graph in P cannot be performed in main memory. The algorithm for computing graphs in P is based on filtering the approximate match table to find all matches associated to Q, and sorting them by their frequencies in ascending order (least frequent first). Then, each of these matches is explored in a breadth-first-search fashion by joining the *M_document* table filtered using *M_predicate* with the transition tables *P_mcall* and *P_ccall* (each explored path is maintained as a tuple in a temporary table). If a valid graph is found, the result is returned to the user. The exploration continues in a recursive manner by joining the resulting table of the previous iteration with the transition tables (those tables that contain the relationships between the methods and classes). The exploration will halt when a certain number of steps has been reached or when a certain time threshold has expired (the exploration can fall into infinite loops). The resulting graphs are returned to the user in ascending order based on the number of elements required to satisfy Q.

6.2.0.9 Experiments

QDPC is a standalone system developed entirely in C# as two modules: a thin client that uses ODBC to connect to the DBMS and a set of UDFs and store procedures that perform the integration and searches. Experiments were run on an Intel Xeon E3110 server at 3.00 GHz with 750 GB of hard drive and 4 GB of RAM. The server was running an instance of SQL SERVER 2005. All the experiments are the result of an average of 30 runs unless otherwise specified. The times are presented in seconds.

Table 6.13: Programs.

Description	WP	SE	PJ1	PJ2
Num. Files	52	44	119	316
Num. Classes	52	0	158	460
Avg. File Size (KB)	1409	4787	12388	11474
Lines of Code (LOC)	5488	5463	28180	70815
Avg. Num. Variables	8	29	19	35
Avg. SQL queries	0	1	0	5
OLTP	N	N	Y	Y

QDPC was tested using four different source code repositories and their corresponding databases' schemas. The programs and schemas are a program used for capturing information to a database of water quality of wells in the State of Texas (WP), the

Table 6.14: Schemas.

Description	WP	SE	PJ1	PJ2
Num. Tables	52	25	21	95
Avg. No. Columns	7	4	6	5
Max No. Columns	110	12	15	29
Min No. Columns	1	2	2	2
Processing Type	OLAP	OLTP	OLTP	OLTP

Sphider open source search engine (SE) program, and a couple of management systems (PJ1 and PJ2). Table 6.13 and Table 6.14 contain the details of each repository and schema.

Figure 6.13 exhibits the results of preprocessing the source code repositories, and obtaining the approximate matches (allowing a proportional edit distance of 10%). This figure also shows that looking for the approximate matches and creating the summarization tables uses only a small portion of the time compared to the source code analysis. Despite this larger source code analysis task, the entire integration phase takes less than 1 minute in all the source code repositories.

Table 6.15 and Figure 6.14 analyze the effect of the approximate matches and the performance of such an exploration. Table 6.15 shows that even though WP contains the fewest LOC, it has the highest number of unique keywords to search. This table shows that the value of β is critical to finding good matches (and avoiding having meaningless matches). As is shown in this table, the values from β should be between 0.10 and 0.25. However, when the value of β exceeds that range, the number of approximate matches increases by a factor of 4x. Furthermore, Figure 6.14 shows that the performance of the algorithm is bounded by the number of unique matches.

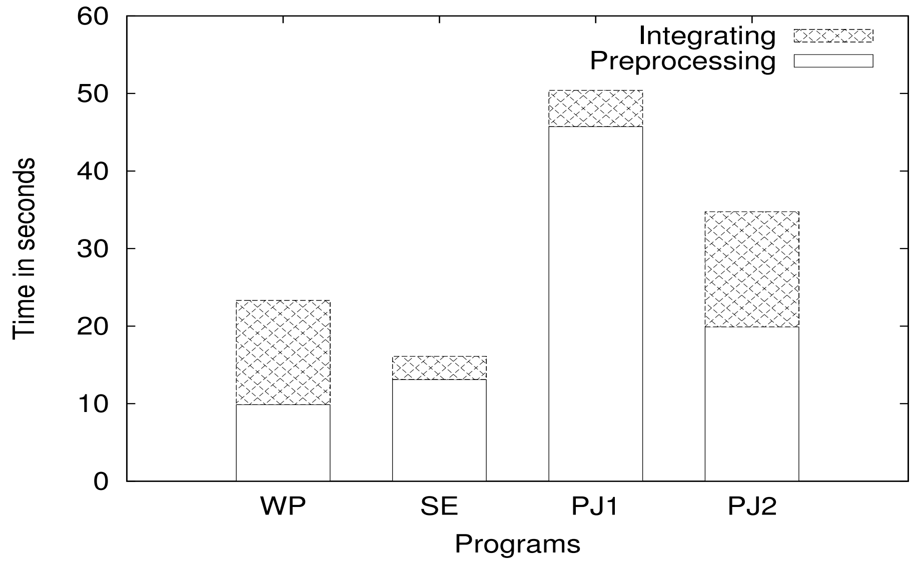


Figure 6.13: Preprocessing and Integration.

PJ2 has the highest performance time due to the number of LOC. However, WP ranks second due to the number of unique keywords to search, regardless of the size of the program. An interesting finding of this plot is that the performance of the algorithm is not affected by the selected β because all of these computations are performed efficiently in batches stored within main memory.

Table 6.15: Integration (Approximate Matches Found.) Performance Time (in seconds) is shown for $\beta = 0.1$.

Program	$M\_predicate$	$\beta = 0.10$	$\beta = 0.25$	$\beta = 0.50$	Time
WP	375	850	979	4008	11
SE	60	1438	1478	3619	4
PJ1	88	4188	4304	13411	5
PJ2	332	42217	42742	143928	16

Table 6.16 shows a breakdown of the performance of the integration phase. As

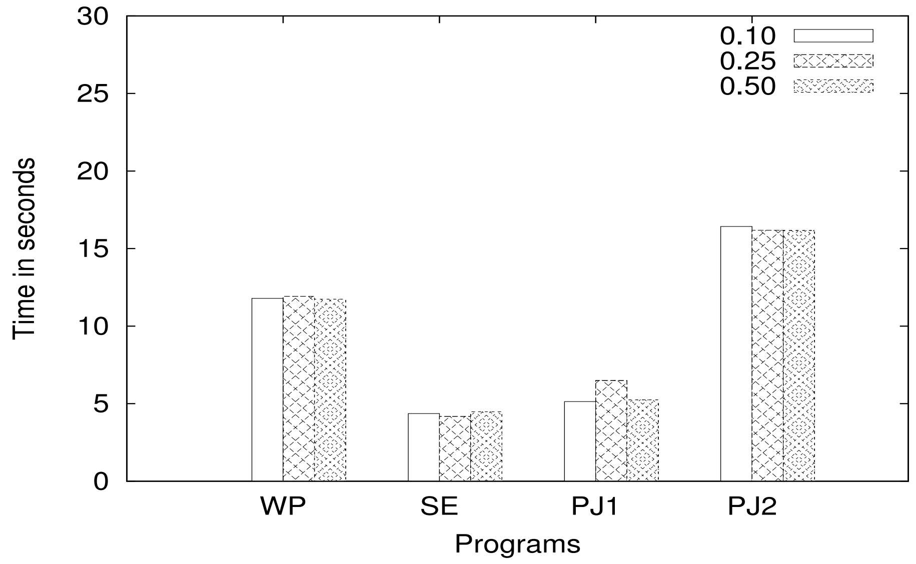


Figure 6.14: Integration (Approximate Matches Performance.)

expected, the bottleneck of the algorithm is in the computation of the approximate matches between the collections (Compute *M_document*).

Table 6.16: Integration (QDPC Profiling.)

Program	Compute M_predicate	Compute M_document
WP	1	4
SE	1	2
PJ1	1	4
PJ2	1	14

The first type of querying to be discussed here involves finding the number of matches associated with elements in either P or S. Figure 6.15 shows the result of obtaining the SUM, MAX, MIN, and COUNT aggregations in a single pass through the data. Figure 6.15 presents similar results for all the collections regardless of the original size. This is due to the fact that the summarization is quite efficient because

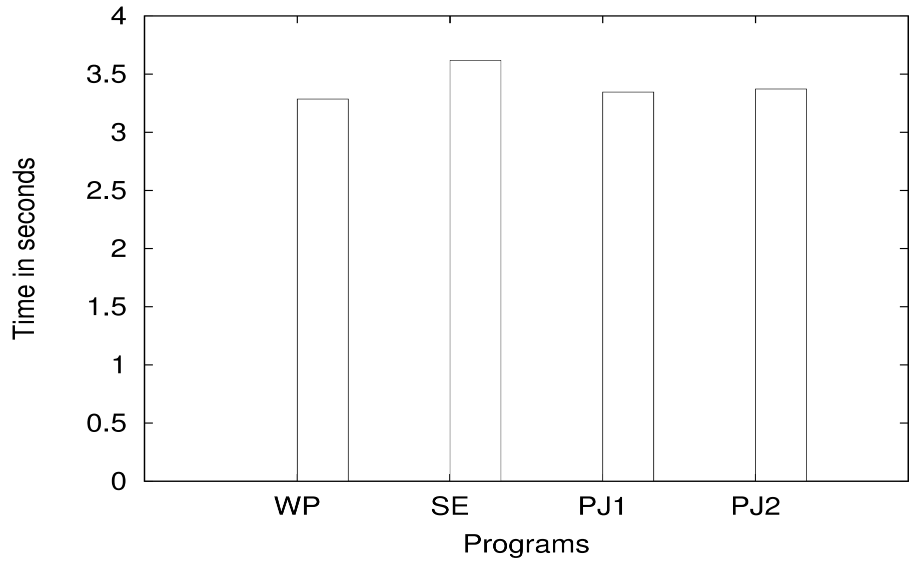


Figure 6.15: Querying (Aggregations.)

the size of the input table is quite small and the aggregate functions are performed within main memory using a hash aggregate. The aggregation is not necessary as long as only these matches are sought, speeding up the query and resulting in a natural join.

A more complex analysis of aggregations can be computed by generating OLAP cubes. These queries answer analytical questions that focus on finding all of the aggregations at different granularity levels. Some of the information obtained in the OLAP cube includes: the matches that are associated with all of the methods and all of the classes, or all the matches that are associated with all of the elements in P. Figures 6.16 and 6.17 show the result of computing an entire OLAP lattice based on the summary tables. The first plot contains the computation of the lattice in different levels of the hierarchy. Therefore, the "P-type, S-type" cube shows

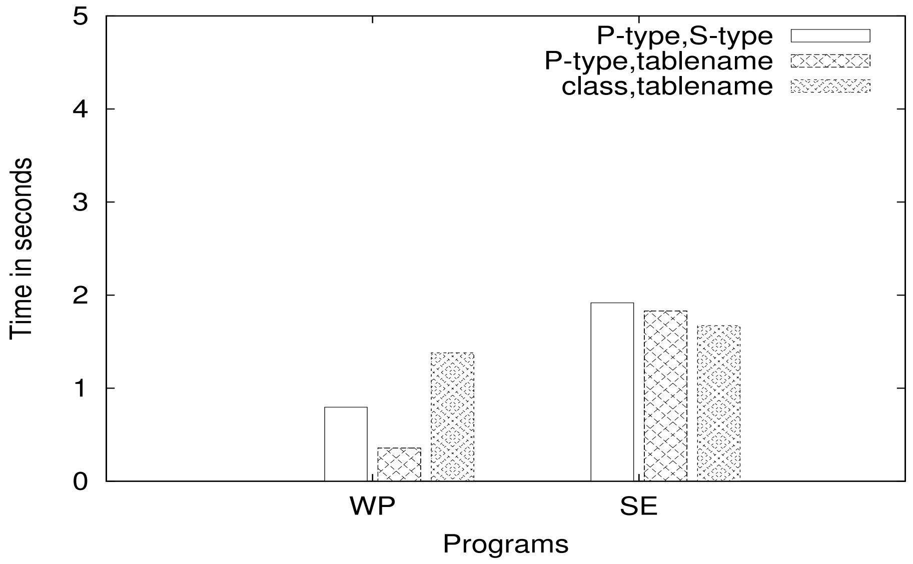


Figure 6.16: Querying (OLAP Cube for Small Projects.)

the number of matches associated with a column, table, class, method, and so on. The "P-type, tablename" obtains an OLAP cube indicating the number of matches associated per v with each T. The last aggregation generates the OLAP cube showing the number of matches associated with every class and T. This plot shows that regardless of the level, the OLAP cube is generated in less than 3 seconds in the smaller programs (see Figure 6.16) and in less than 17 seconds in the larger ones (see Figure 6.17). The highest times are associated with the programs with the largest number of approximate matches. In addition, Figure 6.16 and Figure 6.17 also show that computing a cube in a lower level in the hierarchy is faster due to the early pruning of the match table allowing the exploitation of a hash aggregate.

Finally, all of the elements associated with a particular keyword search are sought, due the dependencies between these. Notice that this type of querying is the most

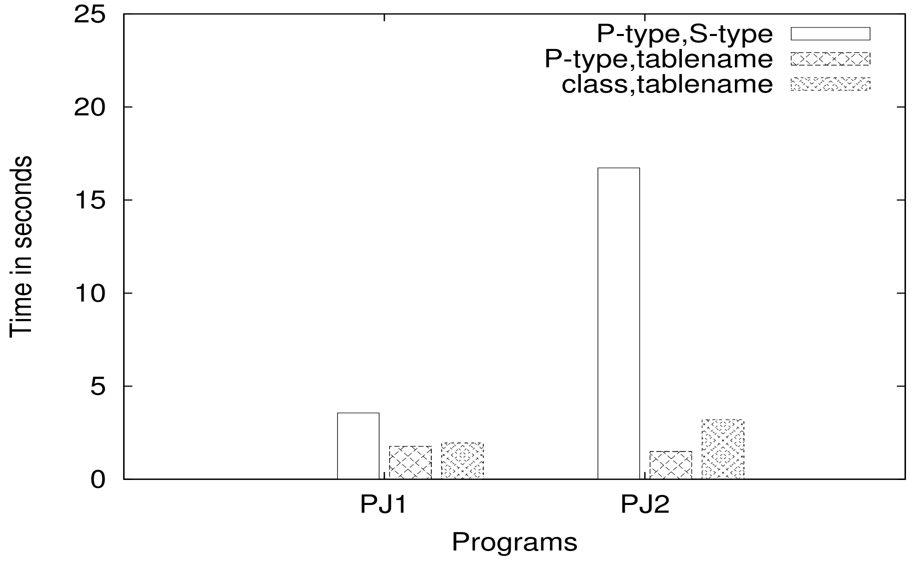


Figure 6.17: Querying (OLAP Cube for Large Projects.)

complex because it requires performing a breadth-first traversal of P and S in order to find all the G that cover a Q. This search answers queries of the form "Is a particular method dependent on a particular column?". Figure 6.18 (which is the result of 30 random conjunctive queries with a maximum recursion depth of 5) shows that the exploration is equally efficient when generating the graphs G by taking only a few seconds for the most time-consuming searches (2 and 3 keywords). When the number of keywords increases in Q, the performance time of the algorithm decreases due to the limited number of graphs that can satisfy Q, resulting in an early termination of the algorithm.

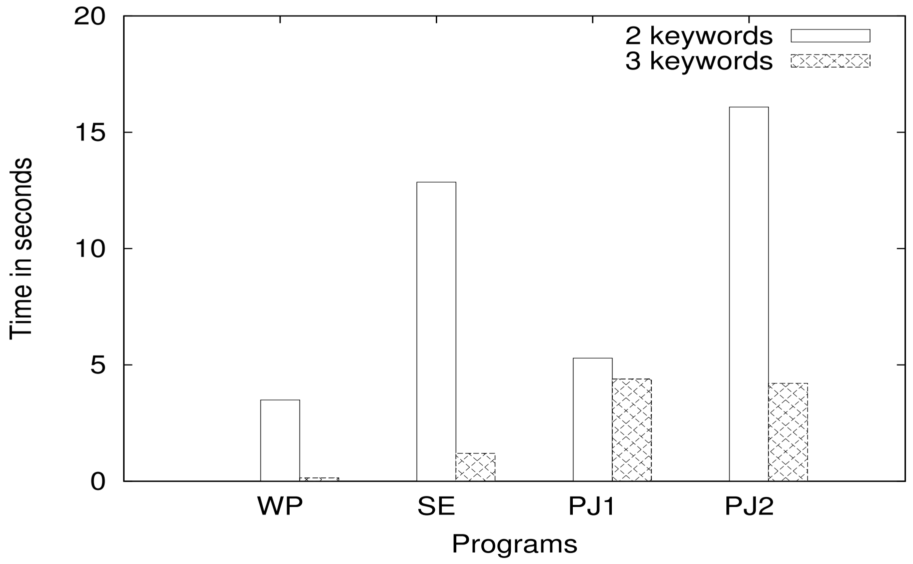


Figure 6.18: Querying (Graph Searches.)

6.3 Conclusions

This chapter introduced Ontocube, an algorithm for ontology extraction with OLAP cubes. The core of the proposal is the obtaining of an efficient keyword frequency summarization with a one-pass algorithm for computing the correlation and lift from pairs of keywords from a corpus. Once this analysis has been performed, a set of heuristics are used to create a hierarchy and find the relationship between the classes. The experiments show that the candidate classes are pruned early with minimal thresholds for lift and correlation to obtain meaningful classes. It was observed that the most influential threshold on the number of classes is lift, with frequency and correlation having similar selectivity effects. It was also found that the most of the classes are quite relevant, and the overall hierarchy accuracy had an overall performance of 67% with a few misclassified concepts decreasing the overall average.

However, for most of the concepts, the hierarchy achieved outstanding results of 75% or higher. The algorithm also exhibited linear scalability with keyword pairing and preprocessing as the most costly steps, respectively.

As a complement to Ontocube, we proposed a novel approach, CUBO, for summarizing text corpora based on a given ontology using OLAP data cubes. CUBO represents an efficient, compact, and scalable data structure and algorithm that generates a data cube for every level of an ontology. Unlike traditional OLAP approaches, CUBO takes advantage of the sparsity of text corpora and generates the combinations for only those dimensions that are present in each document.

All the data cubes are stored in an array with all the levels of a given ontology for efficiency purposes. Every level in the ontology that is analyzed is stored in a hash table and computed for only those existing dimensions in each document. CUBO was adapted into a relational database system using extensibility mechanisms, such as store procedures. The algorithm was tested in real and synthetic databases, and it was shown that the algorithm has linear scalability when the number of documents increases in the collection. In addition, it was shown that unlike traditional algorithms, CUBO is able to scale to a larger number of dimensions due to the avoidance of computing and storing unnecessary combinations. Moreover, the experiments showed that the running time of the algorithm in real and synthetic databases is significantly less than the expected exponential upper bound. This was also appreciated when analyzing the overhead of computing the lattice in higher levels of the hierarchy, resulting in only a small time difference (almost negligible). Hence, the complexity of the algorithm is still highly dependent upon the average number of dimensions in a document. Further analysis of the algorithm in both databases proved that the processing time for building and storing the CUBO is mostly spent on the latter. As a result, CUBO showed to be an efficient and scalable algorithm for summarizing text data with hierarchies.

The final section of this chapter presents an extension to the integration algorithms in the context of source code analysis related to a database schema. The result is a novel system that allows flexible querying of a source code repository and the schema of a database. In order to do so, our approach relies on an efficient integration phase that summarizes these dependencies and stores them in the database management system. The keyword searches in the approach are performed over these summary structures that allow all the graphs that cover the keywords given by the user to be returned. Additional searches based on aggregation and OLAP cubes can be generated using these summary tables to answer complex analytical queries. The experiments also showed that integration of several programs varying in the levels of dependency with the database's schema can be obtained quite efficiently (in less than a minute in all the cases). Furthermore, the scalability of the algorithm is observed to be bounded by the number of keywords to be searched for. It was also shown that searching for dependency graphs that cover the keywords of a query can also be conducted in less than 20 seconds in all of the cases tested. Moreover, the algorithm also prunes all the undesired matches early (if there are not matches covering all the keywords), resulting in a faster evaluation when no graphs are found. The integration obtained with the proposed system also allowed an efficient evaluation of aggregate functions and an efficient construction of OLAP cubes, allowing such complex queries as, “What are the methods that have the highest number of associated dependencies?” to be answered.

6.4 Summary of Contributions

The topics and algorithms presented in this chapter introduce innovative research paths that extend and adapt the integration and OLAP algorithms presented in the previous chapters. This extends the state of the art for exploiting and adapting algorithms that are well known in the database community for analyzing heterogeneous sources.

Ontocube represents an efficient unsupervised algorithm that extracts concepts for text corpora and generates an ontology. The ontology construction is based on some heuristics that establish the relationship between the concepts. To the best of our knowledge, this is the only algorithm that exploits an OLAP cube for building ontologies. Future research includes hybrid approaches (NLP, ML, OLAP), trying other heuristics, and finding other types of relationships between classes.

CUBO is a complement to Ontocube since it takes an ontology and builds a hierarchical data cube. This algorithm is not the first of its kind for building hierarchies inside a DBMS, but it is the only one that takes an ontology and builds a data cube by exploiting the sparse nature of text data. The result of the “on-the-fly” combination computation is that it allows storing a larger number of dimensions than traditional algorithms, since it stores only the combination of dimensions that are present in the data. Future work on this topic includes testing the performance of the algorithm when a new level of indirection is added for capturing instances under the $h-1$ level. Also, we would like to test the algorithm with larger ontologies that do not fit in main memory and have to be loaded in a relational table, as well as more complex real ontologies that have a large depth. Coverage of more complex ontologies that include unbalanced hierarchies and multiple parents is part of future work, too.

The last section of this chapter introduced a novel approach called QDPC, a unique system that allows flexible querying of a source code repository and the schema of a database. The integration presented in QDPC allows the complex computing of efficient aggregations, OLAP queries, and graph querying.

Several problems remain to be addressed in future research. Additional work is required to improve the integration and exploration of source code repositories and a database's schema (e.g. parallel processing techniques). A richer analysis of the semantics of the source code (e.g. considering the type of the data) needs to be incorporated. Moreover, a way to deal with the ambiguity between the source code repository and the schema (e.g. "id" may be a column of several tables) needs to be discovered. The expansion of this method for representing complex relationships and interactions between multiple programs and databases remains to be explored. The optimization for string matching with multiple keywords represents an area of opportunity. Along the line of source code analysis, statistical models (e.g. linear regression) can be introduced to identify which source code elements are more prone to producing bugs. Finally, we believe that it is possible to extend and generalize the proposal to integrate, explore and rank complex programs, databases and documentation efficiently in a database management system.

Chapter 7

Extended Notation for Unstructured Data

This chapter focuses on setting the ground for having unified notation and properties regarding the integration and exploration of structured and unstructured sources.

7.1 Extended Notation

Our unified notation and properties analysis is divided into data preprocessing, data integration, and knowledge discovery phases.

7.1.1 Data Preprocessing

Unstructured data, such as text corpora, possess several properties that make it challenging and difficult to query. Unstructured data do not posses an schema and the relation between the keywords and their relevance is unknown. In addition, the semantics of a particular keyword modify its meaning which could lead to several keywords sharing the same meaning. Based on all these properties, the number of assumptions that can be made about the data are minimal without the support of some domain knowledge. On the other hand, structured data possesses a schema. The data have meaning and the relationships between the data are explicitly defined as functional dependencies, referential integrity, and constraints.

All of the keywords' properties dictate their modeling, as well as the type of operations that can be can performed within them. As defined in the Definition Section 2, let a document d_i contain a string representing a keyword k, where k is a concept that consists of a single word or more words (e.g. 'football field' or 'sports'). Despite the fact the relationships between k's are not defined by a schema, it is possible to extract some structure via the definition of taxonomies. Hence, a concept k' can be a generalization of keyword k''.

A keyword k can be modified by transformation functions f, such as stemmers or soundex functions. Also, a keyword k can be modified by scalar functions that return a measurement between two or more keywords. Finally, the entire data set of k can be transformed or analyzed to create new data sets or measurements that represent the relationships between all of the k's.

Transformation functions f are not commutative, since the order on which these functions are applied returns different results. On the other hand, scalar and analytical functions do not modify the original data and can be applied in a commutative manner.

7.1.2 Data Integration

Following the basic definitions, let us focus on the linkage of both sources. In order to do so, we propose some definitions for obtaining such "links". Intuitively, a link represents a reference of an element in the database and a document or a part of it (e.g. keyword).

Definition 1. *Let $S \supseteq$ all possible links between D and C.*

The universe of all possible links that exist between both sources is contained in S. As such, S contains links that can be inferred from analyzing both collections, which are the focus of this paper, and some other types of links that are the result of previous knowledge of both sources (e.g. a user defined object mapping). Based on the definition of e, which is $e =< k, \gamma >$, let a link between a database element and a document keyword be formally defined as follows:

Definition 2. *Let ρ be a relationship representing a link between a document d_i and an element in the database e based on the finding of approximate keywords of a given element in D.*
$\rho = (e, d_i) \in S | A_k \neq \emptyset.$

Table 7.1: Example of E and L.

L		E	
kid	k	kid	γ
1	student	1	$\{T = student\}$
2	major	2	$\{T = major\}$
...		2	$\{T = student, c = lastname, PK = A890\}$
5	balance	5	$\{T = account, c = balance\}$
6	sid	6	$\{T = transcript, c = SID\}$
...		6	$\{T = account, c = SID\}$
20	a888	20	$\{T = student, c = ID, PK = A888\}$
21	english	20	$\{T = account, c = SID, PK = 503\}$
...		...	

These newly inferred "links" now are identified as relationships between both sources. As a result, an attempt will be made to find all of e that are related to all of the keywords in the documents. Moreover, let E be the set of all of the elements in the database considering the granularity level. Therefore, $E = \{e|e(k,\gamma) \in D\}$. In addition, let L be a set of all unique keywords existing in D without considering γ. Hence, $L = \{distinct(k)|e \in E\}$

For example, "major" exists in D, such that "major" can be found as a table name and as a value in a row in one of the tables in D (e.g. as a surname). In such a particular case, "major" has two entries in E and only one in L. In addition, the "major" keyword may have more references in the record level, therefore linking more records to a keyword. The construction of E is then the result of creating a set of unique keywords L and a dictionary that maps each k to a granularity level. Notice these data structures (see Table 7.1) can be designed and maintained quite easily and efficiently in a database system with tables in 3NF.

Even though E and L define the keywords that are contained in the database, an additional set is required to store the pairs between a keyword and a set of documents. Then, let R store all the $< k, d_i >$ pairs. As a result, $R = \{(k, d_i)|k \in d_i \ and \ d_i \in C\}$. Based on this, the maximum size of R is bounded by the number of unique keywords and the number of documents.

Property 1. *The size of the R data structure is at most $n|L|$, where L contains only the common keywords between D and C.*

Proof. There can only exist a unique pair of $< k, d_i >$ in R given that $A_k \neq \emptyset$ for d_i. Therefore, the maximum number of pairs for a k is the total number of documents in the collection n. Hence, the maximum size of the structure is that every keyword in $|L|$ is contained in every document in the database. $\square$

The e elements in the DBMS must only be extracted only once all the $k \in R$ have been discovered. In addition, ρ is the result of a inner join between the E and R data sets. Notice that the ρ_γ (relationships at each granularity level) are independent sets. As a result, a relationship existing between a keyword of a granularity level in the database may not exist in another granularity level. Despite this, by the transitive properties expressed as part of the Entity-Relation Model (a value in a row is covered by a column c and a table T_j. Similarly, a column c is covered by a table T_j) that exist between the granularity elements in the DBMS (e.g. a table contains attributes and records), one can "assume" new relationships by performing aggregations. A summarization of the sets and structures is show in Table 7.2.

For example, assume a small database with a single $T_1(a_1,a_2)$ and a corpus of a single document $d_1 = \{a_1\}$. The relationships inferred from both sources will

Table 7.2: Data Structures and Subscripts

Index	Range	Used for
i	$1 \dots n$	Documents
j	$1 \dots m$	Tables
l	$1 \dots \lvert L \rvert$	Unique set of k
g	$1 \dots \lvert E \rvert$	Elements in D

be $\rho_1 = (a_1, d_1)$. With further aggregations, one could assume that there is a new relation $\rho_{\mathcal{F}1} = (T_1, d_1)$. These types of relationships will be considered derived, and the finding of these relationships is the result of a posterior analysis of the initial relationships found.

Definition 3. *Let $\rho_{\mathcal{F}}$ be a derived relationship iff $\rho_{\mathcal{F}}$ is equivalent to $\rho = (e_{\gamma_1}, d_i)$ obtained from $\rho' = (e_{\gamma_2}, d_i)$ where $\gamma_1 \geq \gamma_2$ and $\rho \notin R$.*

Furthermore, these derived relationships represent a weaker link between an element in the database and the related document. However, they cannot be ignored. In particular, these new relationships can overlap with already existing links between an element in the DBMS and a document. In addition, notice that one could also obtain some derived relationships by assigning a link between every row of a table given a relationship between a table and a document. However, this type of relationship is considered to be less significant than connecting an element in a higher granularity level. In addition, this important property of a link is exploited during the retrieval of the results as a ranking criterion. As such, derived relationships are always computed from a lower granularity level to a higher one.

These ρ and $\rho_{\mathcal{F}}$ relationships can be used to compute how relevant a keyword is among both collections. Based on this, different traditional ranking techniques, such

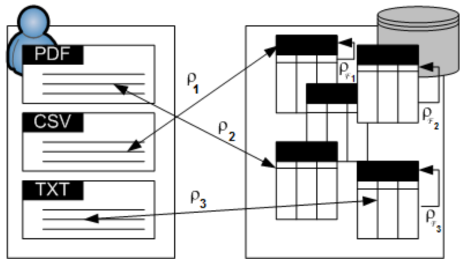


Figure 7.1: Relationships ρ and $\rho_{\mathcal{F}}$ between D and C.

as the Vector Space Model, Probabilistic Models or any other traditional ranking model can be applied to query these documents with additional summarization data structures. Figure 7.1 summarizes the relationship and derived relationships that can be inferred between a database and a set of documents.

7.1.3 Relationship Discovery

The complexity of searching across the inferred relationships is strictly tied to the taxonomy that can be extracted from the keywords k of the unstructured data. As a generalization of the taxonomy-types presented in the definitions, the relationships between unstructured data can be represented as trees, direct acyclic graphs (DAGs) or graphs. The consequence of the "topology" of the unstructured data is that the searches become more costly in terms of nodes that have to be explored.

Let a database D be a balanced tree. In a similar manner, let a set of n keywords in the unstructured domain have different topologies. Lastly, let r represent the number of relationships that are analyzed within the integrated sources. Therefore, based on the topology-type of the unstructured data the following properties can be observed:

Property 2. *The maximum number of elements associated within all the inferred relationships given* r *is of the order of* nr*, since the maximum number of levels of the schema is a constant.*

Property 3. *The maximum number of elements associated within all the inferred relationships in a binary tree structure, given* r*, is of the order of* $r \log n$*, since the logn is the maximum height of the balanced tree.*

Property 4. *The maximum number of elements associated within all the inferred relationships in a general graph, given* r*, is of the order of* nr *with the possibility of having cycles.*

In the first property, there is no possibility for optimizations since nothing can be inferred between the taxonomy of the keywords. The second property is a particular case of a general graph. The advantage of such an approach is that it is possible to avoid visiting other branches that are not related to the given r. The possibility of cycles in a general graph is a problem that has to be tackled during the exploration phase. This can be approached by limiting the number of levels to traverse or 'counting' the number of keywords that were visited.

As can be seen given these properties of unstructured data, if no assumptions or efforts are made for extracting some structure in the unstructured domain, the maximum number of elements that must be explored has the complexity of a "cross product". On the other hand, effective indexing and assumptions about the data (e.g. binary tree) speeds up the searching strategies.

7.2 Conclusions

In this section, we formalizes and provides properties of the structured and unstructured data for exploring the integrated data. Unlike our previous sections, the focus of this section is to summarize the properties of the inherited complexity of the unstructured data, from the preprocessing phase to the searching and knowledge discovery phase. This also shows the properties from the integrated data and the bounds that exists for the data structures explored in chapter 5. In addition, an analysis of the cost that implies exploring integrated sources with different topologies was presented here, as a complement to the exploration shown in chapter 6.

Finally, as it was shown in this chapter, the worst case scenarios results in a costly cross product that requires scanning all the keywords in all the data sets. Therefore future efficient algorithms have to focus on avoiding multiple passes through the data, early pruning of duplicates and the elimination of irrelevant keywords.

7.3 Summary of Contributions

This chapter's main contribution is the analysis and formalization of the algorithms explored during this dissertation, which results in proposal of general properties of integrated structured and unstructured data. Moreover, this chapter also provided storage upper bounds of the data structures presented in this thesis.

However, there are multiple questions that remain to be addressed in future research. Some of these problems are exploring the consequence of adding a semantic meaning to a keyword, which may result in complex relationships that the current proposal may not be able to capture. Also, there is a pressing need to develop benchmarks that can be used to compare the performance of the obtained algorithms. Finally, the extension of SQL to support preprocessing, integrating and querying unstructured data remain as a major challenge to be resolved.

Chapter 8

General Conclusions

This chapter presents a summary of the document's overall contributions, and contains a discussion on this dissertation's general conclusions Finally, there is a brief discussion of the open problems that remain to be explored in the area of document preprocessing inside a DBMS, unstructured data integration, query recommendation, OLAP on unstructured data, and source code analysis.

8.1 Most Important Contributions

This research extends the capabilities of a traditional relational database management system to support unstructured data. In order to do so, several data layouts, data structures and algorithms are proposed to support efficient performance.

The document's proposed algorithms offer efficient in-memory tools that permit the preprocessing of already loaded unstructured data. The advantage of these algorithms is that they reduce the time that it would take to export and process data sets outside the DBMS. In addition, this work presents algorithms that allow flexible query in contrast to those used in ad-hoc system. Furthermore, this dissertation proposes a data layout that reduces the number of I/Os required to retrieve the data, as well as storage tables that can allocate pre-computations for traditional ranking models computed inside a DBMS. An additional contribution is that this data layout supports an extension for the Rank-Join algorithm to be executed within the DBMS as a UDF

A new in-memory UDF-based algorithm for integrating structured and unstructured data is also provided. Finally, these preprocessed or integrated data are exploited with newly proposed UDF-based algorithms for query recommendation and ontology extraction.

Unlike previous approaches, all of the algorithms presented here are contained within the database system, resulting in an entire system that is able to process unstructured data. The system also supports flexible querying for future knowledge discovery algorithms.

The vision of this research extends to uncharted territory by using OLAP cubes for query recommendation and ontology extraction in text corpora. Moreover, the data integration model presented here has opened up a new topic of exploration by allowing the answering of complex queries between a source code repository and the schema of a database.

We have provided efficient algorithms, data layouts, and data structures that support the preprocessing, integrating, querying, and exploring text data with various algorithms that allow knowledge discovery. The contributions of the research, explained by this dissertation, extend a traditional DBMS such that it may manage unstructured data and drive the entire process of transforming the data and discovering new knowledge within it.

The advantages of such a system are evident in the source code exploration section of this dissertation, in which all of these tools are used to explore the source code connected to the schema of a database in an efficient manner. However, there are multiple areas of opportunity that remain to be explored (e.g. medical domain or patent searching). Hence, these tools offer and advantageous unified system for managing unstructured data within a DBMS, opening the door to the exploration of the deep web.

8.2 Future Research

Given the number of topics this dissertation explores, there are multiple areas of investigation that remain open for further research. The following section identify major opportunities left open for exploration based on the areas presented in this dissertation.

Part of future work in document storage and retrieval involves improving the time needed for loading documents into the DBMS, which has become a bottleneck for the implementation of an IR system. The implementation of other storage techniques experimenting with different indexes also remain as possible future investigation. Retrieval proved to be fast enough to consider comparing the results with some benchmarks, but loading and preprocessing large amounts of text data neared the limits of the DBMS's capabilities. Also, it was observed that the optimizations presented here performed better than some other commonly used techniques for building IR-engines in SQL, and we are considering comparing his SQL application with a similar engine, using just user-defined functions. Finally, we would like to explore how the research in a DBMS for implementing text searching models can give acceptable performance in document processing and retrieval by exploiting the database's strengths.

Top-k future work implies implementing different ranking formulas dissimilar to the VSM presented in this dissertation, in order to observe the performance with the Okapi Probabilistic formula or the Dirichlet Prior. Also, the Rank-Join algorithm focuses only on conjunctive queries, and extending this schema to work with disjunctive queries makes it more challenging to guarantee the return of the top-k queries with a pipelining algorithm. Additionally, a much deeper exploration needs to the done on the pipelining algorithm, to guarantee that any top-k result is not lost in the top-k propagation during the execution tree of the algorithm. This is particularly true if the α parameter is large enough to define such a propagation independently from the data distribution. Also, we believe that it is possible to stop the top-k algorithm much earlier in the database scan, and at the same time avoid loading large data sets into memory.

Relationship discovery and structured and unstructured data exploration is possibly the topics covered in this thesis that have a much broader area of improvement and the expansion of ideas. First, defining which relationships are more relevant than others is vital to the final user. Also, a complete analysis of the scalability of relationship generation by experimenting in massive document collections remains unexplored. Another important improvement is to consider the data type and context matching between the database's data and the documents' keywords. At the same time, the expansion of the relationship definition to target a more specific set of records (e.g. adding a PK attribute in atomic relationships) and finding some structure in the unstructured sources may modify the relationship-matching schema presented here. Multicore processing, cloud computing, and parallel computing are other research areas in which the proposed algorithms can be improved upon in order to allow further acceleration in the process presented in this dissertation. Despite the larger numbers of optimizations and explorations that can be done in all of these fields, we believe that this dissertation proves that the DBMS can be exploited for managing structured and unstructured sources efficiently and effectively.

Subsequent goals to finding efficient algorithms for schema matching include extracting a structure from the unstructured sources in order to obtain better "descriptors" and achieve a deeper level of integration (schema mapping). These descriptors can be found by extracting the hidden entities that exist within the unstructured sources. A related problem is resolving ambiguities and data types, which is particularly important when dealing with scientific databases. Therefore, future research plans will focus on resolving entity extraction, ambiguity, and data type resolution.

Taking into consideration the semantics of the data will permit obtaining better quality results when querying and exploring these integrated sources.

Data preprocessing and integration are not complete if it is not possible to return meaningful results to the user. As a consequence, new ranking models, query expansion, query recommendation, and query exploration are part of the topics that remain to be seen.

In particular, future research needs to be conducted to develop new ranking models that can be adapted to the integrated model aside from the proposed VSM. This includes new measurements that can be used as part of the ranking methods to identify the most relevant relationships. Also, the proposal of new exploration algorithms that provide new views of the data are required, such as high dimensionality data cubes is needed. Another opportunity for future work includes extending the OLAP query recommendations to consider new scalable and efficient algorithms for returning suggestions to the user. This should address the current limitation of performing "on-the-fly" summarizations in massive databases. The efficient extraction of statistical measurements is important for the proposal of more robust and efficient algorithms.

Future work could focus on finding efficient, scalable and effective data mining algorithms that will be adapted properly to deal with heterogeneous sources. As a result, exploring possible solutions should include taking advantage of the extensibility functions of a DBMS, MPP databases, and MapReduce. Another unexplored area with huge potential is the extension of using OLAP to create and explore ontologies from heterogeneous.

Finally, the application of the current integration and exploration algorithms may be extended in the future to handle any type of heterogeneous data. However, we believe that the present work has shown that it is possible to extend a relational database management system to offer a unified platform for managing heterogeneous data.

Bibliography

[1] S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti. Scalable ad-hoc entity extraction from text collections. In *Proc. of VLDB Conference*, pages 945–957, 2008.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE Conference*, pages 5–16, 2002.

[3] G. Antoniou and F.v. Harmelen. Web ontology language: OWL. In Peter Bernus, Jacek Blazewics, Gnter Schmidt, Michael Shaw, Steffen Staab, and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 91–110. Springer Berlin Heidelberg, 2009.

[4] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[5] G. Avigdor. Why is schema matching tough and what can we do about it? *ACM SIGMOD Record*, 35(4):2–5, 2006.

[6] T.T. Avrahami, L. Yau, L. Si, and J. Callan. The fedlemur project: Federated search in the real world. *Journal of the American Society for Information Science and Technology*, 57(3):347–358, 2006.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of ACM PODS Conference*, pages 1–16, 2002.

[8] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, and T. Wang. A system for keyword proximity search on XML databases. In *Proc. of VLDB Conference*, pages 1069–1072, 2003.

[9] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *ACM SIGMOD Record*, 28(1):54–59, 1999.

[10] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. of IEEE ICDE Conference*, pages 431–440, 2002.

[11] M. Bhide, V. Chakravarthy, A. Gupta, H. Gupta, M. Mohania, K. Puniyani, P. Roy, S. Roy, and V. Sengar. Enhanced Business Intelligence using EROCS. In *Proc. of IEEE ICDE Conference*, pages 1616–1619, 2008.

[12] M.A. Bhide, A. Gupta, R. Gupta, P. Roy, M.K. Mohania, and Z. Ichhaporia. LIPTUS: associating structured and unstructured information in a banking environment. In *Proc. of ACM SIGMOD Conference*, pages 915–924, 2007.

[13] J.A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman. .NET database programmability and extensibility in Microsoft SQL Server. In *Proc. of ACM SIGMOD Conference*, pages 1087–1098, 2008.

[14] P. Buitelaar, D. Olejnik, and M. Sintek. OntoLT: A Protégé plug-in for ontology extraction from text. In *Proc. of ISWC*, pages 3–4, 2003.

[15] V.T. Chakaravarthy, H. Gupta, P. Roy, and M. Mohania. Efficiently linking text documents with relevant structured information. In *Proc. of VLDB Conference*, pages 667–678, 2006.

[16] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping. In *Proc. of CIDR Conference*, pages 1–12, 2005.

[17] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *Proc. of ACM SIGMOD Conference*, pages 1005–1010, 2009.

[18] Z. Chen, C. Garcia-Alvarado, and C. Ordonez. Enhancing document exploration with olap. In *Proc. of IEEE ICDM Conference*, pages 1407–1410, 2010.

[19] Z. Chen and C. Ordonez. Efficient OLAP with UDFs. In *Proc. of ACM CIKM Workshop on Data Warehousing and OLAP (DOLAP)*, pages 41–48, 2008.

[20] Z. Chen, C. Ordonez, and C. Garcia-Alvarado. Fast and dynamic OLAP exploration using UDFs. In *Proc. of ACM SIGMOD Conference*, pages 1087–1090, 2009.

[21] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *Proc. of VLDB Conference*, pages 1481–1492, 2009.

[22] W.B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley, 2010.

[23] J. Driscoll D.A. Grossman. Structuring text within a relational system. In *Proc. of International Conference on Database and Expert Systems Applications (DEXA)*, pages 72–77, 1992.

[24] M.Y. Dahab, H.A. Hassan, and A. Rafea. TextOntoEx: Automatic ontology construction from natural English text. *Journal on Expert Systems with Applications*, 34(2):1474–1480, 2008.

[25] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proc. of VLDB Conference*, pages 451–462, 2006.

[26] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *Proc. of Conference on Innovative Data Systems Research*, pages 9–12, 2011.

[27] H.H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. *Lecture Notes in Computer Science*, pages 221–237, 2003.

[28] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 4th edition, 2003.

[29] A. Elsayed, S.R. El-Beltagy, M. Rafea, and O. Hegazy. Applying Data Mining for Ontology Building. In *Proc. of IEEE International Workshop on Security in e-Science and e-Research (ISSR)*, pages 1:1–1:14, 2007.

[30] R. Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[31] R. Fagin, A. Lotem., and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of ACM PODS Conference*, pages 102–113, 2001.

[32] H. Fang, T. Tao, and C.-X. Zhai. A formal study of information retrieval heuristics. In *Proc. of ACM SIGIR Conference*, pages 49–56, 2004.

[33] B.M. Fonseca, P.B. Golgher, E.S. De Moura, and N. Ziviani. Using association rules to discover search engines related queries. In *LA-WEB*, pages 66–71, 2003.

[34] N. Fukuta, T. Yamaguchi, T. Morita, and N. Izumi. DODDLE-OWL: Interactive Domain Ontology Development with Open Source Software in Java. *IEICE TOIS*, 91(4):945–958, 2008.

[35] A. Gal, G. Modica, and H. Jamil. Ontobuilder: Fully automatic extraction and consolidation of ontologies from web sources. In *Proc. of ICDE*, page 853, 2004.

[36] C. Garcia-Alvarado, Z. Chen, and C. Ordonez. OLAP with UDFs in digital libraries. In *Proc. of ACM CIKM Conference*, pages 2073–2074, 2009.

[37] C. Garcia-Alvarado, Z. Chen, and C. Ordonez. OLAP-based query recommendation. In *Proc. of ACM CIKM Conference*, pages 1353–1356, 2010.

[38] C. Garcia-Alvarado, Z. Chen, and C. Ordonez. ONTOCUBE: Efficient ontology extraction using OLAP cubes. In *Proc. of ACM CIKM Conference*, pages 2429–2432, 2011.

[39] C. Garcia-Alvarado and C. Ordonez. Information retrieval from digital libraries in SQL. In *Proc. of ACM CIKM Workshop on Web Information and Data Management (WIDM)*, pages 55–62, 2008.

[40] C. Garcia-Alvarado and C. Ordonez. Keyword search across databases and documents. In *Proc. of ACM SIGMOD Workshop on Keyword Search on Structured Data (KEYS)*, pages 2:1–2:6, 2010.

[41] C. Garcia-Alvarado and C. Ordonez. Integrating and querying web databases and documents. In *Proc. of ACM CIKM Conference*, pages 2369–2372, 2011.

[42] C. Garcia-Alvarado and C. Ordonez. Integrating and querying source code of programs working on a database. In *Proc. of ACM SIGMOD Workshop on Keyword Search on Structured Data (KEYS)*, pages 47–53, 2012.

[43] C. Garcia-Alvarado and C. Ordonez. Query processing on cubes mapped from ontologies to dimension hierarchies. In *Proc. of ACM CIKM Workshop on Data Warehousing and OLAP (DOLAP)*, pages 57–64, 2012.

[44] C. Garcia-Alvarado, C. Ordonez, and V. Baladandayuthapani. Querying external source code files of programs connecting to a relational database. In *Proc. of ACM Ph.D. Workshop in Information and Knowledge Management (PIKM, CIKM Conference Workshop)*, pages 9–16, 2012.

[45] C. Garcia-Alvarado, C. Ordonez, and Z. Chen. DBDOC: Querying and browsing databases and interrelated documents. In *Proc. of ACM SIGMOD Workshop on Keyword Search on Structured Data (KEYS)*, pages 47–48, 2009.

[46] C. Garcia-Alvarado, C. Ordonez, and Z. Chen. Query recommendation in digital libraries using OLAP. In *Proc. of ACM SIGMOD Workshop on Keyword Search on Structured Data (KEYS)*, pages 6:1–6:2, 2010.

[47] C. Garcia-Alvarado, V. Raghavan, S. Narayanan, and M.F. Waas. Automatic data placement in mpp databases. In *Proc. of IEEE ICDE Workshop on Self-Managing Database Systems (SMDB)*, pages 322–327, 2012.

[48] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[49] M.N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look: A tutorial. In *Proc. of ACM SIGMOD Conference*, page 635, 2002.

[50] A. Giacometti, P. Marcel, E. Negre, and A. Soulet. Query recommendations for OLAP discovery driven analysis. In *Proc. of ACM CIKM Workshop on Data Warehousing and OLAP (DOLAP)*, pages 81–88, 2009.

[51] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of VLDB Conference*, pages 26–37, 1998.

[52] T. Grabs, K. Böhm, and H.J. Schek. PowerDB-IR: information retrieval on top of a database cluster. In *Proc. of ACM CIKM Conference*, pages 411–418, 2001.

[53] T. Grabs, K. Böhm, and H.J. Schek. PowerDB-IR–Scalable Information Retrieval and Storage with a Cluster of Databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[54] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. In *Proc. of IEEE ICDE Conference*, pages 152–159, 1996.

[55] D.A. Grossman and O. Frieder. *Information Retrieval: Algorithms And Heuristics*. Springer, 2004.

[56] D.A. Grossman, D.O. Holmes, and O. Frieder. A parallel DBMS approach to IR in TREC-3. In *Proc. of TREC*, pages 279–279, 1994.

[57] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *Proc. of ACM SIGMOD Conference*, pages 16–27, 2003.

[58] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: the teenage years. In *Proc. of VLDB Conference*, pages 9–16, 2006.

[59] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, San Francisco, 1st edition, 2001.

[60] J. Henrard and J.L. Hainaut. Data dependency elicitation in database reverse engineering. In *Proc of IEEE European Conference on Software Maintenance and Reengineering*, pages 11–19, 2001.

[61] D. Holmes. SQL text parsing for information retrieval. In *Proc. of ACM CIKM*, pages 496–499, 2003.

[62] I. Horrocks. DAML+ OIL: a description logic for the semantic web. *IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, 2002.

[63] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proc. of VLDB Conference*, pages 850–861, 2003.

[64] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. *ACM SIGMOD Record*, 30(2):259–270, 2001.

[65] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proc. of VLDB Conference*, pages 670–681, 2002.

[66] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 13(1):49–70, 2004.

[67] S. Hwang and K.C. Chang. Optimizing top-k queries for middleware access: A unified cost-based approach. *ACM Transactions on Database Systems (TODS)*, 32(1):5, 2007.

[68] I.F. Ilyas, W.G. Aref, and A.K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[69] I.F. Ilyas, G. Beskales, and M.A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, pages 1–58, 2008.

[70] A.R. Jaiswal, C.L. Giles, P. Mitra, and J.Z. Wang. An architecture for creating collaborative semantically capable scientific data sharing infrastructures. In *Proc. of ACM CIKM Workshop on Web Information and Data Management (WIDM)*, pages 75–82, 2006.

[71] N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(4):525–539, 2006.

[72] G. Koutrika, Z.M. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. In *Proc. of EDBT Conference*, pages 391–402, 2009.

[73] J. Lee, D. Grossman, O. Frieder, and M.C. McCabe. Integrating structured data and text: A multi-dimensional approach. In *Proc. of IEEE International Conference on Information Technology: Coding and Computing*, pages 264–269, 2000.

[74] G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proc. of ACM SIGMOD Conference*, pages 903–914, 2008.

[75] G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *Proc. of IEEE ICDE Conference*, pages 1183–1186, 2009.

[76] W.S. Li and C. Clifton. Semantic integration in heterogeneous databases using neural networks. In *Proc. of VLDB Conference*, page 1, 1994.

[77] X. Li, J. Han, Z. Yin, J.G. Lee, and Y. Sun. Sampling cube: A framework for statistical OLAP over sampling data. In *Proc. of ACM SIGMOD Conference*, pages 779–790, 2008.

[78] C.X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao. Text cube: Computing IR measures for multidimensional text database analysis. In *Proc. of IEEE ICDM Conference*, pages 905–910, 2008.

[79] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. of ACM SIGMOD Conference*, pages 563–574, 2006.

[80] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proc. of ACM SIGMOD Conference*, pages 115–126, 2007.

[81] S. Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, 2012.

[82] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Proc. of VLDB Conference*, pages 49–58, 2001.

[83] M.C. McCabe, D. Holmes, D.A. Grossman, and O. Frieder. Parallel platform-independent implementation of information retrieval algorithms. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1:1–1:6, 2000.

[84] A.K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/ mccallum/bow, 1996.

[85] A.K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[86] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of VLDB Conference*, pages 122–133, 1998.

[87] A. Natsev, Y.C. Chang, J.R. Smith, C.S. Li, and J.S. Vitter. Supporting incremental join queries on ranked inputs. In *Proc. of VLDB Conference*, pages 281–290, 2001.

[88] C. Ordonez. Optimizing recursive queries in SQL. In *Proc. of ACM SIGMOD Conference*, pages 834–839, 2005.

[89] C. Ordonez. Statistical Model Computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[90] C. Ordonez. Data set preprocessing and transformation in a database system. *Intelligent Data Analysis (IDA)*, 15(4):613–631, 2011.

[91] C. Ordonez, Z. Chen, and J. García-García. Metadata management for federated databases. In *ACM CIMS Workshop*, pages 31–38, 2007.

[92] C. Ordonez and C. Garcia-Alvarado. A data mining system based on sql queries and udfs for relational databases. In *Proc. of ACM CIKM Conference*, pages 2521–2524, 2011.

[93] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems Journal*, 44(2):495–508, 2008.

[94] C. Ordonez, J. Garcia-Garcia, R. Montero-Campos, and C. Garcia-Alvarado. A referential integrity browser for distributed databases. In *Proc. of ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 14:1–14:2, 2009.

[95] C. Ordonez, N. Mohanam, C. Garcia-Alvarado, P.T. Tosic, and E. Martinez. Fast PCA Computation in a DBMS with Aggregate UDFs and LAPACK. In *Proc. of ACM CIKM Conference*, pages 2219–2223, 2012.

[96] C. Ordonez, M. Navas, and C. Garcia-Alvarado. Parallel multithreaded processing for data set summarization on multicore CPUs. *Journal of Computing Science and Engineering (JCSE)*, 5(2):111–120, 2011.

[97] C. Ordonez, I.Y. Song, and C. Garcia-Alvarado. Relational versus non-relational database systems for data warehousing. In *Proc. of ACM CIKM Workshop on Data Warehousing and OLAP (DOLAP)*, pages 67–68, 2010.

[98] C.D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.

[99] M. Papiani, J.L. Wason, A.N. Dunlop, and D.A. Nicole. A distributed scientific data archive using the Web, XML and SQL/MED. *SIGMOD Record*, 28(3):56–62, 1999.

[100] J.M. Perez, R. Berlanga, M.J. Aramburu, and T.B.Pedersen. R-cubes: OLAP cubes contextualized with documents. In *Proc. of IEEE ICDE Conference*, pages 1477–1478, 2007.

[101] S. Pitchaimalai, C. Ordonez, and C. Garcia-Alvarado. Efficient distance computation using SQL queries and UDFs. In *Proc. of IEEE HPDM*, pages 533–542, 2008.

[102] S. Pitchaimalai, C. Ordonez, and C. Garcia-Alvarado. Comparing SQL and MapReduce to compute Naive Bayes in a single table scan. In *Proc. of ACM CloudDB*, pages 9–16, 2010.

[103] L. Qin, J.X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *Proc. of ACM SIGMOD Conference*, pages 681–694, 2009.

[104] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[105] L. Salmela, J. Tarhio, and P.Kalsi. Approximate Boyer-Moore string matching for small alphabets. *Algorithmica*, 58(3):591–609, 2009.

[106] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[107] M. Sayyadian, H. LeKhac, A.H. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *Proc. of IEEE ICDE Conference*, pages 346–355, 2007.

[108] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Lecture Notes in Computer Science*, 3730(1):146–171, 2005.

[109] A. Simitsis, G. Koutrika, and Y. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB Journal*, 17(1):117–149, 2008.

[110] J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. In *Proc. of Lecture Notes in Computer Science*, pages 348–359. Springer, 1990.

[111] P. Vassiliadis. A survey of extract-transform-load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[112] Q.H. Vu, B.C. Ooi, D. Papadias, and A.K.H. Tung. A graph method for keyword-based selection of the top-K databases. In *Proc. of ACM SIGMOD Conference*, pages 915–926, 2008.

[113] G. Weikum. DB&IR: both sides now. In *Proc. of ACM SIGMOD Conference*, pages 25–30, 2007.

[114] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proc. of ACM SIGMOD Conference*, pages 527–538, 2005.

[115] N. Ye. *The Handbook of Data Mining.* Lawrence Erlbaum, 2003.

[116] D. Zhang, C. Zhai, and J. Han. Topic cube: Topic modeling for olap on multidimensional text databases. In *Proc. of SIAM SDM Conference*, pages 1124–1135, 2009.

[117] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.