Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Υπολογιστων και Πληροφορικης

Μοντελοποίηση και θέματα λειτουργίας βάσεων προτύπων

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

του

ΜΑΝΩΛΗ ΤΕΡΡΟΒΙΤΗ

Διπλωματούχου Ηλεκτρολόγου Μηχανικού
Μηχανικού Υπολογιστών Ε.Μ.Π. (2001)

Αθήνα, 14η Μαΐου 2007

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Υπολογιστων και Πληροφορικης

Μοντελοποίηση και θέματα λειτουργίας βάσεων προτύπων

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

του

ΜΑΝΩΛΗ ΤΕΡΡΟΒΙΤΗ

Διπλωματούχου Ηλεκτρολόγου Μηχανικού
Μηχανικού Υπολογιστών Ε.Μ.Π. (2001)

Συμβουλευτική Επιτροπή: Τ. Σελλής
Ι. Βασιλείου
Ε. Ζάχος

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή, 14η Μαΐου 2007.

...	...	...
Τ. Σελλής	Ι. Βασιλείου	Ε. Ζάχος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.
...	...	...
Φ. Αφράτη	Κ. Σαγώνας	Ι. Θεοδωρίδης
Καθηγητής Ε.Μ.Π.	Αναπ. Καθ. Ε.Μ.Π.	Επίκ. Καθ. ΠΑ.ΠΕΙ.
	...	
	Σ. Σκιαδόπουλος	
	Επικ. Καθ. ΠΑ.ΠΕ.	

Αθήνα, 14η Μαΐου 2007

...

Μανώλης Τερροβίτης
Δρ. Ηλεκτρολόγος Μηχ.

National Technical University of Athens

School of Electrical and
Computer Engineering

Computer Science Division

Modelling and Operational Issues for Pattern Base Management Systems

PhD Thesis

of

Manolis Terrovitis

Diploma in Electrical Engineering (2001)

Athens, 14th May 2007

National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division

Modelling and Operational Issues for Pattern Base Management Systems

PhD Thesis

of

Manolis Terrovitis

Diploma in Electrical Engineering (2001)

Supervising Committee: T. Sellis
Y. Vassiliou
E. Zachos

Approved by the Examination Committee, 14th May 2007.

...	...	...
T. Sellis	Y. Vassiliou	E. Zachos
Prof. NTUA	Prof. NTUA	Prof. NTUA
...	...	...
F. Afrati	K. Sagonas	Y. Theodoridis
Prof. NTUA	Associate Prof. NTUA	Assistant Prof. UNIPI
	...	
	S. Skiadopoulos	
	Assistant Professor UOP	

Athens, 14th May 2007

...

Manolis Terrovitis
Electrical Engineer, PhD

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Databases both as a research and an application field have already several mature characteristics by computer science standards. They have an established role in production, a fairly stable conceptual and logical model behind their applications and they effectively address their primary objective: managing huge volumes of data. Still, in the recent years the requirements have changed substantially as basic database technology is employed in several new fields, like biomedical data, data in peer-to-peer systems, data from sensor networks etc. Moreover, it is the very notion of management of data that is changing; storing and selecting even with complex criteria does not seem enough to compensate for the needs of nowadays applications. The users that handle huge volumes of data want to explore them efficiently and view results in a concise way; often they want to view results that have not been described meticulously at query time.

Complex analysis for data is usually carried out by specialized tools that do not adhere to some standard. The most general and systematic method for analyzing data in depth has been carried out in the data mining field. Cluster identification, construction of decision trees, frequent itemset analysis and others, comprise the most recognized methods in data mining. Such functionality, often appears coupled with database management in several commercial DBMSs. Still, the smooth incorporation of data mining and other forms of complex analysis in the database management is far from becoming a reality.

Most research efforts to incorporate data mining functionality in databases, focus on the algorithmic part of the challenge; how to make data mining algorithms as transparent and as easy to use as any query operator. Whereas this is an important problem, we believe that a more interesting challenge lies in the incorporation of *the management of the patterns* that are produced by data mining methods, in traditional RDBMSs. Instead of focusing in reconciliating data mining methods with databases, we focus on the management of their results. This approach comprises an important research challenge for several reasons:

- *It is essential.* Both database research and commercial database systems have identified the need to bring data mining and RDBMSs closer. Pattern management is the first essential step in such an effort. It comes natural that if we want to have complex analysis methods in traditional databases, we need to be able to store and query their results. Most data mining methods usually produce vast amounts of patterns, which need to be further processed in order to identify the most interesting ones.

- *It is interesting.* Storing and querying patterns that come from different data mining/analysis methods enables the user to explore data in-depth. Patterns in this context, express some quality/property of the data. The ability to combine them, or test them in new datasets, or simply compare them, allows the user to identify interesting properties of her/his datasets that are not visible by just browsing them. For example, being able to trace quickly which of the data that belong to a certain cluster contain some specific frequent itemsets, might reveal an interesting subset of the data.

- *It is feasible.* Whereas there have been several efforts in providing data mining functionality in RDBMSs by introducing new query operators that support some data mining method, we are still far from incorporating data mining in databases. One of the reasons is the complexity and the variety of the different data mining algorithms. Despite the large variety of algorithms and methods, the different kinds of the produced patterns are significantly more limited. This comprises an important motivation for focusing firstly on the handling of the patterns.

- *It is difficult to solve.* The fact that pattern management is promising does not make it a trivial issue. A series of research challenges arise both in modeling and implementing a pattern management system. The core of the problems lies in the rich semantics of the patterns, and more specifically in modeling and exploiting their relation with the raw data.

The work in this thesis is a first step towards creating a unified system for managing both patterns and data. We aim at creating a system where the user will be able to query both patterns and data to facilitate the discovery of interesting qualities of the dataset. We propose a system that handles both data and patterns as primary citizens amongst its entities, the *Pattern Warehouse*. The pattern warehouse, allows the user to further exploit the results of data mining by combining existing patterns to create new ones, by providing the tools to reason about the qualities of a dataset based on the related patterns, and by letting the user to test the patterns against different datasets. We have identified four basic targets in our effort:

1. The user must be able to pose queries against both data and patterns by using a common query language. At the same time, we want patterns to keep their complex semantics that distinct them from raw data.

2. We want a transparent representation of pattern semantics, which facilitates the creation of some reasoning mechanism and allows the users to combine them to create new patterns.

3. We need to retain the relation between data and patterns, and make this information available to the queries. The end-user must be able to navigate easily from the pattern to the data space and vice versa.

4. The implementation challenges, especially those regarding the navigation between patterns and data must be answered efficiently.

The goals set for the Pattern Warehouse sketch a huge volume of research work, which cannot be addressed in full extent in this thesis. In the scope of this work we attack the problem from two opposing ends: we present a conceptual and a logical model for incorporating the handling of patterns in a DBMS, and on the other end, we propose methods for extending current database technology for evaluating containment queries. The former approach provides a model for a system that handles both patterns and data and provides insight on how the user can exploit this knowledge to quickly explore huge data volumes. The latter approach works at the opposite end of the problem; it provides more efficient solutions to the navigation between the pattern and the data space. In addition, as our research on modelling issues demonstrated, any integration of raw data with data patterns will rely heavily on being able to quickly decide on membership properties for various data entities and on the relations between large datasets. This is the same problem we face in current database systems, when we want to evaluate containment queries. Current database research has partly overlooked the problem of containment evaluation, since it usually assumed the existence of a collection of strictly structured tuples and rarely of a collection of *set-values*.

In short the basic contributions of this thesis are as follows:

- We present a conceptual model for the Pattern Warehouse, which is an environment that offers all the functionality of a modern DBMS, but in addition allows for handling *patterns*. In this scope, we consider patterns as condensed forms of knowledge artifacts that are derived from various methods of data analysis. Common examples are clusters, frequent itemsets, etc.

- We propose a logical model for the Pattern Warehouse, which treats both patterns and data as primary citizens and allows for creating a common query

language for both entities. The model facilitates the transparent representation of the internal structure of the patterns and of their relation with the underlying data.

- We provide a sound and complete model of pattern relations that support some basic reasoning capabilities of the pattern warehouse. These relations are used to define a series of operators that give insight on the functionality of the Pattern Warehouse.

- We attack the problem of the navigation between the data and the pattern space by focusing of the efficient evaluation of containment queries. Our results have a broader range of applications, including containment evaluation in current DBMSs. We describe in detail the problem of containment evaluation using the current state-of-the-art index, the inverted file, and we highlight its weaknesses. Based on this analysis we propose a novel index, the Hybrid Trie-Inverted file (HTI) index.

- We propose a second mechanism for speeding up containment queries, the *Ordered Inverted File* (OIF) index, which relies on pre-ordering the set-values. This solution offers better results in environments where the updates are infrequent.

1.1 Thesis organization

The conceptual and logical modeling issues of the pattern warehouse are covered in Chapters 2 and 3. Chapter 3 concludes with the introduction of queries that facilitate the navigation between the pattern and the data space. Indexing methods for enhancing such a functionality are presented in the next chapters, which address the generalized problem of containment queries. In more detail, the thesis is structured as follows:

- *Chapter 2.* In this chapter we present the framework for the pattern warehouse. We present the conceptual architecture of a three-layered environment covering both data and patterns and we complement it with the required logical model. Moreover, we present a method for describing the relation between patterns and data in a concise manner.

- *Chapter 3.* In this chapter we define a complete and sound set of binary pattern predicates/relations and we present a series of operators. The operators concern querying and updating the pattern warehouse and provide a first glimpse of its functionality.

- *Chapter 4.* In Chapter 4 we present the basic set of containment queries that are in the focus of our research and the most popular solutions for their evaluation. We emphasize the description of inverted files due to their superior performance and widespread application. We show how queries are evaluated and the most efficient techniques for creating and maintaining and inverted file. We analyze its strengths and weaknesses to open way for the new proposals that follow in Chapters 5 and 6.

- *Chapter 5.* We present our proposal for handling skewed distributions, the HTI index. HTI relies on breaking up the longest inverted lists to smaller sub-lists, which contain known combinations of the most frequent items of the database. In this chapter we detail HTI's structure and we show how the trie can be used to prune large parts of the search space during query evaluation. We also present algorithms for creating and maintaining the HTI in the presence of updates.

- *Chapter 6.* In Chapter 6, we complement our proposal for containment query evaluation with one more index, the OIF. OIF relies on sorting the set-values before indexing them. As a result of this sorting, queries can effectively prune parts of the search space independently of the items' distribution. Again we present the index in detail and we explain how all types of queries are evaluated. We complement the work with a complexity analysis for the query evaluation and a detailed description of the creation and maintenance algorithms.

- *Chapter 7.* Finally, we conclude the thesis by summarizing the basic research results and by presenting issues for future work.

Chapter 2

A Logical Model For Pattern Warehouses

2.1 Introduction

Nowadays, we are experiencing a phenomenon of information overload, which escalates beyond any of our traditional beliefs. As a recent survey states [LV00], the world produces between 1 and 2 exabytes of unique data per year, 90% of which is digital and with a 50% annual growth rate. Clearly, this sheer volume of collected data in digital form calls for novel information extraction, management, and querying techniques, thus posing the need for novel Database Management Systems (DBMSs). Still, even novel DBMS architectures are insufficient to cover the gap between the exponential growth of data and the slow growth of our understanding [Gra02], due to our methodological bottlenecks and simple human limitations.

In the previous decade, data warehousing was introduced to compensate for the difficulties to support any decision making over vast volumes of raw data. Data warehouses provide On-Line Analytical Processing (OLAP) capabilities to the end-user. OLAP processing comprises a set of tools that allow for exploring and querying data, organized and aggregated in different levels of granularity. For example, a business expert can browse the statistics of a company organized and aggregated by year, region etc. Still, as both the size and the complexity of the data continues to grow, simple aggregations fall short in addressing current needs for data exploration and decision making.

It is often the case that complicated semi-automatic specialized procedures are needed to explore huge datasets and discover the desired knowledge. The most profound example of such procedures are the various data mining techniques. Data mining methods (frequent itemset discovery, clustering, decision trees) create knowledge artifacts from the underlying raw data, by following a semi-automatic procedure.

The results of these methods are a lot more flexible in describing the properties of the underlying dataset, than simple aggregations provided by the data warehouse. We term these knowledge artifacts as *patterns*, and we consider them as compact representations of the knowledge hidden in the data. In other words, we consider the creation of patterns as a reduction of the available data, (through data processing methods, pattern recognition, data mining, knowledge extraction) to a more compact form, that preserves as much as possible from their hidden/interesting/available information.

Data mining analysis is usually carried out by specialized tools that do not adhere to some standard. The extracted patterns are usually modelled and stored in an arbitrary format. No emphasis is given in further querying and processing them. Still, it is often the case that most of the data analysis methods produce a plethora of results, which need to be further investigated in order to identify the interesting ones. A prominent way of addressing this problem is to employ methods and techniques from traditional DBMSs to handle the patterns. If patterns are stored in a DBMS, then we can exploit all the powerful indexing, storing and querying tools that are available to facilitate our investigations. Nevertheless, despite the fact that data mining functionality, often appears coupled with database management in several commercial DBMSs, the smooth incorporation of data mining and other forms of complex analysis in the database management is far from becoming a reality.

So far, patterns have not been adequately treated as *persistent objects* that can be stored, retrieved, and queried. Thus, the challenge of integration between patterns and data seems to be achievable by designing fundamental approaches for providing database support to pattern management. In particular, since patterns represent relevant knowledge, often very large in volume, it is important that such knowledge is handled as *first-class* citizen. This means that *patterns should be modelled, stored, processed, and queried in a similar way to data in traditional DBMSs.* As a first step towards this goal we propose the Pattern Warehouse, which encompasses a both a pattern base and a database. The proposed model addresses patterns arising in a variety of data analysis methods, ranging from signal processing to data mining. In short our contributions in this chapter are:

- First, we describe the requirements for pattern management, and based on our analysis we present the conceptual architecture for the Pattern Warehouse, an environment that incorporates both pattern and data management utilities. We describe its layers and the entities that reside in it, and we demonstrate its capabilities by a motivating example.

- Second, we define the foundations for the global setting of pattern management through a logical model that covers data, patterns and intermediate mappings.

The model covers typing issues, both for data and patterns, with the latter provision aiming towards being able to define reusable pattern types that are customized per case. Moreover, the model allows the database designer to organize semantically similar patterns in the appropriate collections that can be subsequently queried, just like relations in traditional databases.

- Finally, we propose two mechanisms for modelling the relation between patterns and data. An explicit one; where each patterns is explicitly linked to the underlying data, through pointer-based mappings, and an implicit one. The latter is a formalism which approximates the relation between the patterns and the data space. To this end we present a declarative formula that covers a certain variety of patterns and complements the precise mappings.

Based on the results of this chapter we complement our work in chapter 3, by introducing a sound and complete model of pattern relations and a series of query operators.

This chapter is organized as follows: in Section 2.2, we present the basic Pattern Warehouse architecture. We continue in Section 2.3 by presenting an example of pattern management and in Section 2.4 we provide the definitions of the logical foundations of the pattern warehouse. Finally, we conclude the chapter in Section 2.5, by providing a short summary of this chapter together with our conclusions.

2.2 The Pattern Warehouse

The main idea behind pattern management is that we can use patterns as abstractions of large datasets in order to explore the properties of the data more efficiently. In our proposal patterns and data are stored and manipulated in a unified environment, the pattern warehouse through a common interface. Nevertheless, we consider patterns as data distinct entities, which are linked by the pattern extraction methods (e.g. data mining). We focus on patterns stemming from data mining procedures, still, we use a broad definition about patterns [RBC+03]:

Definition 2.2.1. *Patterns are compact and rich in semantics representations of raw data.*

Patterns serve as artifacts, which describe (a subset of) data with similar properties and/or behavior, providing a compact and rich in semantics representation of data. This definition allows for a very general selection of knowledge artifacts that can be considered as patterns. We envisage a system that could handle patterns stemming from various application areas, like signal processing, image recognition, sensor monitoring, etc. We aim to provide an efficient way of exploring complex

qualities of very large datasets. Based on this vision for data processing we have identified a series of requirements as guide in our research effort, which are presented in the next Section.

2.2.1 Requirement Analysis

The requirements we have gathered come both from the application experience of experts (mainly in the context of the PANDA project), but also as a result of our understanding of the limitations of data mining. They are formed around the main axis of the idea of a unified environment for handling both patterns and data. In more detail, we expect that such an environment should satisfy the following requirements:

- Both patterns and data should be managed in a unified environment, through a common interface. Queries can be executed against pattern or data classes, or even both. Despite the common interface we consider patterns and data distinct entities.

- The user should be able to navigate efficiently from the pattern to the data space, back and forth. Patterns are related to their data images and data are mapped to patterns corresponding to them.

- Patterns and their relationships with the raw data are expressed in a generic, extensible and transparent way. The logical representation of the patterns allows the easy extension with new types of patterns, and facilitates the creation of patterns from existing ones.

- Patterns can exist independently of the underlying data. If a class of patterns has been derived from a large dataset, it is not compulsory to have access to the dataset in order to manage the patterns.

- A pattern management system should allow the user to avoid performing costly operations over large data spaces, but instead she should be able to decide upon qualities of the dataset by using its concise pattern representations. This implies that a variety of query operators for patterns and patterns classes should be available. We expect that functionality like selecting patterns by value, comparing them to each other or matching for similarity or equivalence will be available as it is for simple data records.

Of course we still have the standard requirements for transaction management, efficient storage, safety in queries etc, as in the case of the DBMSs. We detail on the ones that stem from the novelty of our approach.

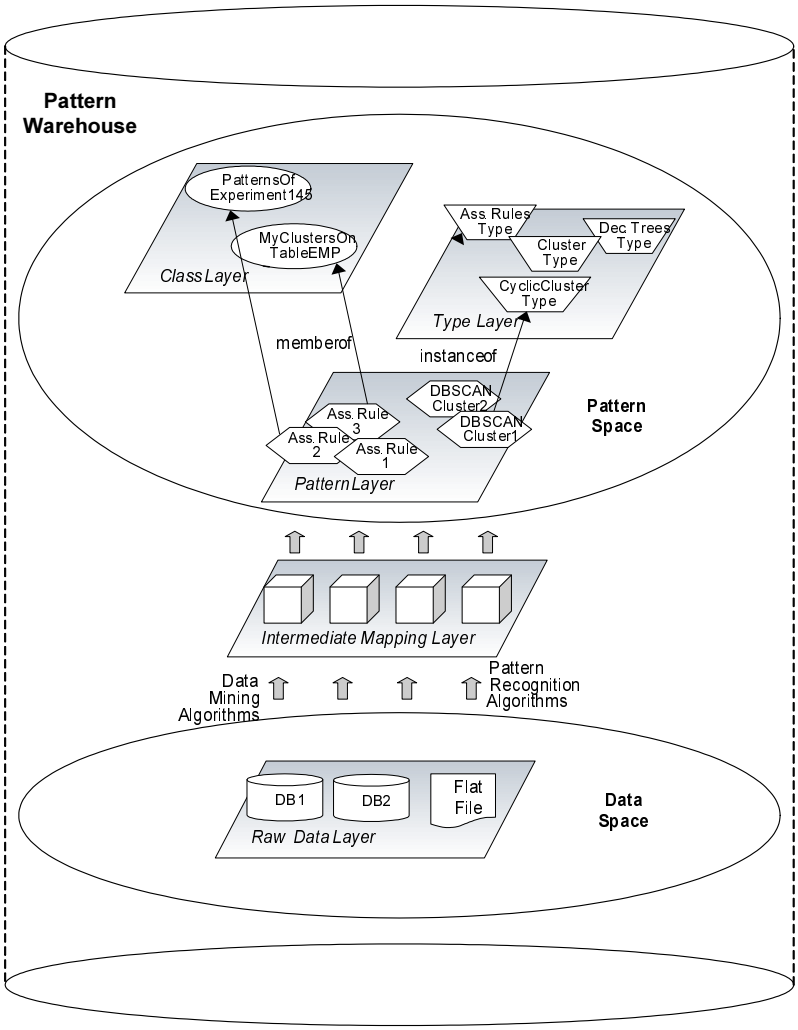


Figure 2.1: *Reference architecture for the Pattern Warehouse*

2.2.2 Conceptual Architecture of a Pattern Warehouse

In order to address the aforementioned requirements we envision an environment where the patterns are managed exactly as database records are managed by a database management system. The reference architecture for such a system is depicted in Fig. 2.1 and consists of three major layers of information organization. In the bottom of Fig. 2.1, we depict the *data space* consisting of data stores that contain data (*data layer*). At the top of Fig. 2.1, we depict the *pattern space* where the patterns reside. Finally, in the middle of Fig. 2.1, we can observe the *intermediate mappings* that relate patterns to their corresponding data, forming the *intermediate data layer*. Intermediate mappings facilitate the justification of any knowledge inferred at the pattern space with respect to the data; for example, they could be used to retrieve the rows that produced the association rules of Fig. 2.1. The overall architecture is the conceptual architecture of the integrated environment termed *Pattern Warehouse* (*PW*).

Next, we present a brief description of all the layers that appear in this abstract

description of the Pattern Warehouse:

- *Data layer*. The data from which the patterns are created reside in this layer. Data can be physically managed by a different repository from the patterns. The motivation for the introduction of the original data in the Pattern Warehouse is dual: (a) there are several queries that require data as their answers, and, (b) from a modeling point of view, data are the measure of correctness for any approximations that occur in the PW setting. Naturally, in a practical setting, data can be absent, or not available to the PW; this can prohibit the answering of several questions, but cannot block the entire functionality of the PW. For reasons of simplicity and uniformity, we assume that the data are organized in an object-relational database.

- *Intermediate mappings layer*. This layer manages information on the relations between the patterns and the underlying data, involving specialized storage and indexing structures. Keeping intermediate mappings is clearly motivated by the existence of cross-over queries, where the user navigates back and forth between the pattern and the data space. The information kept here comprises catalogs linking each pattern with the data it represents. In general, one can imagine that these catalogs are huge and might not be available for every pattern, or they might be very expensive to be queried often. To address this problem, we consider two approaches for the representation of information about the pattern-data relation: (a) *exact*, where intermediate mappings are described by catalogs; and (b) *approximate*, where a formula is used to describe the pattern in the data space. In the latter case, one can employ different variations of these approximations through data reduction techniques (e.g., wavelets), summaries, or even on-line data mining algorithms. We emphasize that the exact intermediate layer can also be unavailable to the PW; the latter can still operate due to the approximations of the patterns' formulae.

- *Pattern layer*. Patterns are the actual information stored in the PW. They are concise and rich in semantics representations of the underlying data. Patterns are the stored information through which a user interacts with the system and performs queries and serve exactly as records serve in a traditional database. Some -but not obligatorily all- patterns are related to the underlying data through intermediate mappings and/or are characterized by an approximation formula.

  This layer is populated by patterns, and it is linked to the data layer through the intermediate mappings. The queries can navigate from this layer to the data layer by using the intermediate mappings. Each pattern of this layer has an image in the data layer.

- *Type layer.* The *PW* pattern types describe the intensional definition, i.e., the syntax of the patterns. Patterns of the same type share similar characteristics, therefore pattern types play the role of data types in traditional DBMS's or object types in OODBMS's. Normally, we anticipate the *PW* to include a predefined set of built-in, popular pattern types (e.g., association rules and clusters). Still, the type layer must be extensible, simply because the set of pattern types that it incorporates must be extensible.

- *Class layer.* The *PW* classes are collections of patterns which share some semantic similarity. Patterns that are members of the same class are required to belong to the same type. Classes are used to store patterns with predefined semantics given by the designer; by doing so, the designer makes it easier for users to work on classes. For example, a class may comprise patterns that resulted from the same experiments, like the association rules in Fig. 2.1. In other words, classes have the same role as tables in relational databases and object classes in object-oriented databases: they are logical-level entrypoints, collecting semantically similar data and easying the formulation of queries for the user.

We use the term *representation* for the relation between the patterns and data, to express our intention to use patterns as *abstractions* of certain properties of data. For example, a frequent itemset captures the fact that certain objects frequently appear together in the underlying set of data. We aim at using the patterns to directly manipulate properties that characterize subsets of the data or even areas of the data space. Although patterns as abstract representations of data are our primary concern, data as supportive evidence of patterns are also of value. Therefore, we need to store the interactions between patterns and their generative data.

The notion of representation and the notion of image come, for reasons that will be made obvious in the rest of the chapter, in two flavors: an exact one, defined with reference to data which are explicitly mapped to the pattern and an approximate one. The latter case expresses the compact nature of a pattern.

Using patterns as representations of data facilitates inferring properties of sets of data by performing operations solely in the pattern space, and vice versa: it allows to resolve queries concerning relations between patterns in the data space, by examining the relations of the represented data sets. **Moreover, keeping the patterns and the data distinct allows for testing whether the property described by a pattern holds in different data sets**. This enables users to discover interesting knowledge, already recorded for some data set, in other data sets without performing again expensive data mining operations.

2.2.3 Novelty of the approach

Why is the problem hard to solve with the current state-of-practice? Assuming that someone would like to treat patterns as first-class citizens (i.e., store and query them), a straightforward approach would be to employ standard object-relational or XML database technology to store patterns and exploit its query facilities to navigate between the two spaces and pose queries. Although it is feasible to come up with an implementation of this kind, several problems occur. A preliminary object-relational implementation was pursued in PANDA [PAN02] and we briefly list our findings here. First, we need to address the problem of having a generic structure able to accommodate as many kinds of patterns as possible. Unfortunately, standard relational and even object-relational technology results in a schema which is too complicated. As typically happens with relational technology, such a complex structure has a direct overhead in the formulation of user queries, which become too lengthy and complicated for the user to construct. An XML database could probably handle the heterogeneity problem gracefully; nevertheless, the main strength of XML databases lies in the efficient answering of path queries. Unfortunately, the particularities of the (pointer-like) navigation between the data and the pattern space as well as the novel query operations concerning collections of patterns are not compatible with this nature of XML path queries.

Based on the above, the situation calls for usage of customized techniques. In particular, we need both a logical model for the concise definition of a pattern base and an appropriate query language for the efficient formulation of queries. As we shall see, in this thesis we treat the problem from a blank sheet of paper and discuss the management of patterns as a sui-generis problem. It is also interesting to point out that having an appropriate logical model, allows its underlying support by an object-relational or an XML storage engine in a way that is transparent to the user.

Why is the problem novel? Both the standardization and the research community have made efforts close to the main problem that we investigate. Nevertheless, to the best of our knowledge there is no approach that successfully covers all the parameters of the problem in a coherent setting.

So far, *standardization efforts* [PMM03, SQL01, JDM03] have focused on providing a common format for describing patterns with the aim of interchangeability. The storage and eventual querying of such patterns has not been taken into consideration in the design of these approaches. A much deeper approach, similar to our own, has been performed in the field of *inductive databases*: a particular characteristic of inductive databases is that the querying process treats data and patterns equally. Nevertheless, the inductive database community avoids the provision of a generic model, capable of handling any kind of patterns and focuses, instead, on

specific categories of them. Therefore, our approaches are complementary since the inductive database community follows a bottom-up approach, with individual cases of patterns considered separately, whereas we follow a top-down approach with a generic model that aims to integrate any kind of patterns. Finally, it is interesting to point out the relationship to the *data warehousing* approach, where data are aggregated and stored in application-oriented data marts. We follow a similar, yet broader, approach, in the sense that although data warehousing aggregation is one possible way to provide semantically rich information to the user in a concise way, it is not the only one. As already mentioned, we try to pursue this goal in an extensible fashion, for a significantly broader range of patterns.

2.3 Motivating example

To show the benefits of treating patterns as first class citizens, we briefly discuss a scenario involving a manager interacting with a set of stored patterns that he can query through the facilities offered by a *Pattern Warehouse*. We will assume the existence of data for the customers and the transactions of two different supermarket branches organized in an object-relational database like the following:

```
CustBranch1(id:int,name:varchar,age:int,income:int,sex:int)
CustBranch2(id:int,name:varchar,age:int,income:int,sex:int)
TransBranch1(id:long int,customerID:int,totalSum:euro,items:{int})
TransBranch2(id:long int,customerID:int,totalSum:euro,items:{int})
```

The relations `CustBranch1`, `CustBranch2` hold data for customers from two different branches of the supermarket and the relations `TransBranch1`, `TransBranch2` hold data for the transactions performed at the respective branches. The scenario takes place during two different days (Fig. 2.2).

Day 1: A data mining algorithm is executed over the underlying data and the results are stored in the pattern base. For example, the data mining algorithm involves the identification of frequent itemsets. Possible patterns stored in the pattern base involve sets of items that are frequently bought together; e.g., {`bread,butter`}. The user is notified on this event and starts interacting with the stored data and patterns. During the early stages of this interaction the user is navigating back and forth from the pattern to data space, in order to accommodate queries like the following ones:

Which are the data represented by a pattern? The user selects an interesting pattern and decides to find out what are the underlying data that relate to it.

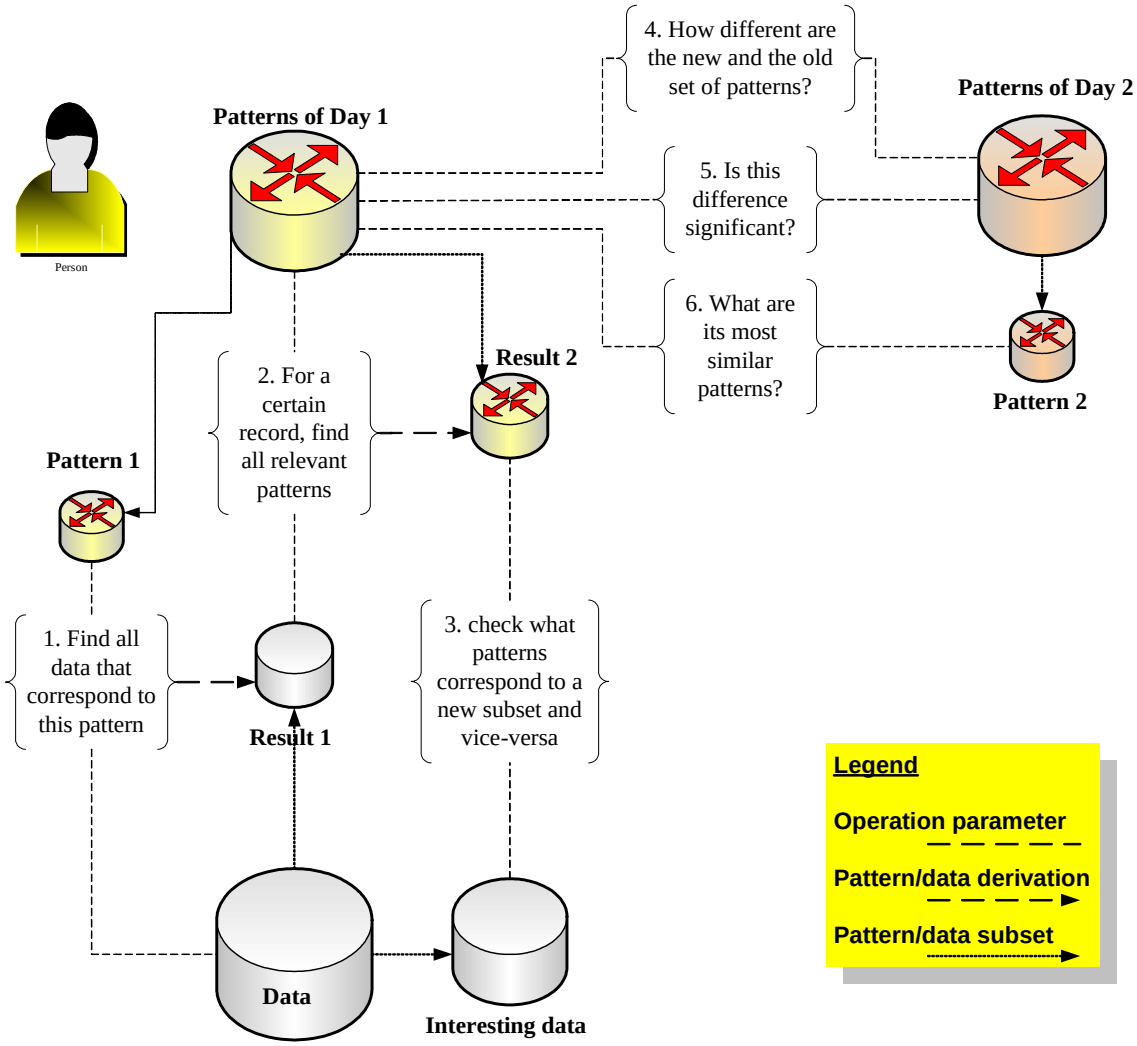


Figure 2.2: *An example of pattern generation scenario*

Which are the patterns representing a data item? Out of a vast number of data that the user receives as an answer, he picks an interesting record and decides to see which other patterns relate to it (so that he can relate the originating pattern with these ones, if this is possible).

Does a pattern represent adequately a data set? This process of "cross-over" operations, like the two mentioned above, goes on for a while, as the user gets accustomed to the data. Then, he decides to proceed to some more elaborate analysis, consisting of "what-if" operations. Therefore, he arbitrarily chooses another dataset (on the basis of some assumption) and checks which of the available patterns can adequately represent it. Then, on the basis of the results, he selects a certain pattern and tries to determine which sets of data correspond to this pattern. This interaction with the stored data and patterns goes on for some time.

Could we derive new patterns from knowledge in the pattern base? The user may exploit patterns stored in the pattern base in order to derive or even conjecture the existence of new patterns representing existing data without applying any time-consuming mining technique but using available query operators.

Day 2: Data in the source databases change, therefore, the original data mining algorithms are executed on a regular basis (e.g., weekly) in order to capture different trends on the data. Once a second version of the patterns is computed (for the new data this time), the analyst is interested in determining the changes that have taken place.

How different are two pattern classes? First, the user wants to find out which are the new patterns and which patterns are missing from the new pattern set. This involves a simple qualitative criterion: presence or absence of a pattern.

Is the difference of two pattern classes significant? Then, the user wants some quantitative picture of the pattern set: which patterns have significant differences in their measures compared to their previous version?

How similar are two patterns? Finally, the user wants to see trends: how could he predict that something is about to happen? So, he picks a new pattern and tries to find its most similar pattern in the previous pattern set, through a similarity operator.

2.4 A logical model for pattern bases

In this section, we give the formal foundations for the treatment of data and patterns within the unifying framework of a pattern warehouse. First, we define the data space according to the complex value model of [AB95], further assuming that data are organized as an object-relational database. Then, we formally define pattern types, pattern classes, patterns, and the intermediate mappings between data and patterns. Finally, we define pattern bases and pattern warehouses.

2.4.1 The data space

Whereas in the general case, data can be organized under many different models without being incompatible to our framework, we need to assume a certain form of organization in order to give the formal logical definitions for the internal structure of a PW. Our choice for the data model is the complex value model, which can be considered as a quite broad variant of the object-relational model [AB95]. This model generalizes the relational model by allowing set and tuple constructors to be recursively applied. We believe that this model is flexible and, at the same time, elegantly extended by user defined predicates and functions. In this section we present, sometimes verbatim, some basic definitions from [AB95] to make the chapter clearer and more self contained.

Data types, in the complex value model adopted, are structured types that use domain names, *set* and *tuple* constructors, and attributes. For simplicity, throughout the rest of this section, our examples consider only integer and real numbers, as well as strings.

Definition 2.4.1. *Data types (or simply,* types*) are defined as follows:*

- *If* $\widehat{D}$ *is a domain name, then* $\widehat{D}$ *is an* atomic *type.*
- *If* $T_1, \ldots, T_n$ *are types and* $A_1, \ldots, A_n$ *are distinct attribute names then* $[A_1{:}T_1, \ldots, A_n{:}T_n]$ *is an unnamed* tuple *type.*
- *If* T *is a type then* $\{T\}$ *is an unnamed* set *type.*
- *If* T *is an unnamed type and* A *is an attribute name, then* $A{:}T$ *is a* named *type.*

The *values* of a specific type are defined in the natural way. For atomic types, we assume an infinite and countable set of values as their domain, which we call **dom**(T). The domain of set types is defined as the powerset of the domain of the composing type. The domain of tuple types is defined as the product of their constituent types.

Example 2.4.1. *Consider the following types.*

$$T_1 = [X{:}real,\ Y{:}real,\ Z{:}integer]$$
$$T_2 = \{\ [Id{:}integer,\ Disk{:}\{\ [Center{:}[X{:}real, Y{:}real],\ Rad{:}real]\ \}\ ]\ \}$$

The expressions

$$[X{:}4.1,\ Y{:}5.2,\ Z{:}3] \text{ and } \{\ [Id{:}7,\ Disk{:}\{\ [Center{:}[X{:}2.0, Y{:}3.0], Rad{:}4.1]\ \}]\ \}$$

are values of types T_1 and T_2 respectively.

Relations in our setting are sets of tuples defined over a certain composite data type, as in the nested relational model. This definition is more restrictive than the one presented in [AB95], which allows for relations that are sets of sets, but here, we adopt it for the sake of simplicity. Also, for each relation, we assume an implicit, system-generated row identifier *RID* that uniquely identifies each tuple. We will not explicitly refer to *RID*'s in the sequel, unless necessary. In this context, a database and a database schema are defined as follows [AB95]:

Definition 2.4.2. *A database schema is a pair $\widehat{DB} = \langle[\widehat{D_1}, \ldots, \widehat{D_k}], [\widehat{R_1}{:}T_1, \ldots, \widehat{R_n}{:}T_n]\rangle$ where $T_1, \ldots, T_n$ are set types involving only the domains $\widehat{D_1}, \ldots, \widehat{D_k}$ and $\widehat{R_1}, \ldots, \widehat{R_n}$ are relation names.*

Definition 2.4.3. *An instance of $\widehat{DB}$, i.e., a database, is a pair $DB = \langle[D_1, \ldots, D_k], [R_1, \ldots, R_n]\rangle$, where R_i's are relations and D_i's are domains.*

We also refer to $\widehat{DB}$ as the *database type*, and to DB as the *database value*.

As we will see in the following, we need to be able to define patterns over joins, projections, and selections over database relations. To this end, we extend Definition 2.4.2 with a set of *materialized views* $V_1, \ldots, V_m$ defined over relations $R_1, \ldots, R_n$ using relational algebra. Throughout the rest of the chapter, we address views as relations, unless explicitly specified otherwise.

2.4.2 The pattern space

In Section 2.2, we have described patterns as compact and semantically rich representations of data and presented their functionalities and relations with the underlying data. In this section, we refine their description and give the logical definition of the entities that reside in pattern, pattern type and pattern class layers. As already mentioned, pattern types are templates for the actual patterns and pattern classes are groups of semantically similar patterns.

Patterns as previously described are formally translated as quintuples. We intuitively discuss the components of a pattern here and give the definition next.

- First, a pattern is uniquely identified by a *Pattern Id* (*PID*).
- Second, a pattern has a *structure*. For example, an association rule comprises a *head* and a *body*, and a circular cluster comprises a *center* and a *radius*.
- Third, a pattern is typically generated by applying a data mining algorithm over some underlying data. By overloading some traditional database terminology, we call this data collection the *active domain* of the pattern. Note that the domain deals with a broader collection of data than the ones that precisely map to the pattern; therefore, the relationship deals with the context of validity of the pattern and not with its exact image in the data space.
- Fourth, a pattern informs the user about its quality, i.e., how closely it approximates reality compared to the underlying data, through a set of statistical *measures*. For example, an association rule is characterized by its *confidence* and *support*.
- Finally, a *mapping formula* provides the semantics of the pattern. This component provides a possibly simplified representation of the relation between data represented by the pattern and the pattern structure. In Section 2.4.5, we present the formalism that we have adopted for expressing this relation.

A *pattern type* represents the intensional description of a pattern, the same way data types represent the intentional description of data in the case of object-relational data. In other words, a pattern type acts as a template for the generation of patterns. Each pattern is an *instance* of a specific pattern type. There are four major components that a pattern type specifies:

- First, the pattern type dictates the structure of its instances through a *structure schema*. For example, it specifies that association rules must comprise a head and a body.
- Moreover, a pattern type specifies the *domain*. The domain describes the schema of the underlying data that is used to generate the patterns. Practically, the domain is the schema of the relations that can be used as input for the pattern detection algorithm.
- Third, it provides a *measure schema*, i.e., a set of statistical measures that quantify the quality of the approximation that is employed by the instances of the pattern type.
- Finally, it provides the mapping formula in terms of data types to provide the general guidelines that connect the underlying data with the pattern to be instantiated by the pattern type. This formula is instantiated in each pattern.

Definition 2.4.4. *A* pattern type *is a quintuple* $[Name, SS, D, MS, MF]$ *where (a)* $Name$ *is a identifier unique among pattern types; (b)* SS*, the* Structure Schema*, is a distinct complex type; (c)* D*, the* Domain*, is a set type; (d)* MS*, the* Measure Schema*, is a tuple of atomic types; and (e)* MF*, the* Mapping Formula*, is a predicate over* SS *and* D*.*

Definition 2.4.5. *A* pattern (instance) p *of a pattern type* PT *is a quintuple* $[PID, S, AD, M, MF]$ *such that (a)* PID *is a unique identifier among all patterns, (b)* S*, the* Structure*, and* M*,* Measure*, are valid values of the respective structure and measure schemata of* PT*, (c)* AD*, the* Active Domain*, is a relation, which instantiates the set type of the Domain, and (d)* MF*, the* mapping formula*, is a predicate expressed in terms of* AD *and* S*, instantiating the respective mapping formula of* PT*.*

In order to define the mapping formula in the pattern type, we need to be able to reference the subelements of the complex value type appearing in the domain field. Since a pattern type is a generic construct not particularly bounded to a specific data set with specified attribute names, we need to employ auxiliary names for the complex value type, that will be overridden in the instantiation procedure.

Let us consider the instantiation procedure that generates patterns based on pattern types. Assume that a certain pattern type PT is instantiated in a new pattern p. Then:

- The data types specified in the Structure Schema SS and the Measure Schema MS of PT are instantiated by valid values in p.
- The auxiliary relation and attribute names in the domain D of PT are replaced by regular relation and attribute names from an underlying database, thus specifying the active domain of the pattern.
- Both the previous instantiations also apply for the Mapping Formula MF. The attributes of the Structure Schema are instantiated to values and the auxiliary names used in the definition of the domain are replaced by the actual relations and attribute names appearing in the active domain.

Having defined the data space and the pattern entities, we are ready to define the notions of *pattern class* and *pattern base (PB)*. Our final goal is to introduce the global framework, called *pattern warehouse*, as a unified environment in the context of which data- and pattern-bases coexist.

A pattern class over a pattern type is a collection of semantically related patterns, which are instances of this particular pattern type. Pattern classes play the role of pattern collections, just like relations are collections of tuples in the relational model.

CustBranch1(id,name,age,income,sex)		CustBranch2(id,name,age,income,sex)	
Cluster 1	Cluster 2	Cluster 3	Cluster 4
[346,A,30,33,1]	[533,E,43,60,1]	[532,I,31,34,1]	[012,O,41,59,1]
[733,B,31,31,2]	[657,F,47,60,2]	[322,J,32,31,2]	[230,N,45,59,1]
[289,C,29,29,1]	[135,G,45,59,2]	[315,K,30,29,0]	[292,M,43,58,1]
[923,D,30,27,2]	[014,I,49,61,2]	[943,H,31,28,1]	[971,L,43,60,1]

Table 2.1: *Clustering in relations* `CustBranch1` *and* `CustBranch2`

Definition 2.4.6. *A* pattern class *is a triplet* $[Name, PT, Extension]$ *such that:(a)* $Name$ *is a unique identifier among all classes; (b)* PT *is a pattern type; and (c)* $Extension$ *is a finite set of patterns with pattern type* PT*.*

Now, we can introduce pattern bases as finite collections of classes defined over a set of pattern types and containing pattern instances.

Definition 2.4.7. *A* Pattern Base Schema *defined over a database schema* $\widehat{DD}$ *is defined as* $\widehat{PB} = \langle[\widehat{D_1}, \ldots, \widehat{D_n}], [\widehat{PC_1}{:}PT_1, \ldots, \widehat{PC_m}{:}PT_m]\rangle$*, where* PT_i*'s are pattern types involving the domains* $\widehat{D_1}, \ldots, \widehat{D_n}$ *and* $\widehat{PC_i}$*'s are pattern class names.*

Definition 2.4.8. *An instance of* $\widehat{PB}$*, i.e., a* pattern base*, over a database* DB *is defined as* $PB = \langle[PT_1, \ldots, PT_k], [PC_1, \ldots, PC_m]\rangle$*, where* PC_i*'s are pattern classes defined over pattern types* PT_i*, with patterns whose data range over the data in* DB*.*

2.4.3 Motivating example revisited

Coming back to our motivating example, assume now that the manager of the company wants to analyze the data about the employees and the products sold and employs a clustering algorithm on the `age` and `income` attributes to find groups of employees that have similar age and salaries. The clustering algorithm returns the clusters depicted in Table 2.1. The tuples with `id` $\in \{346, 733, 289, 923\}$ are represented by Cluster 1, the tuples with `id` $\in \{533, 657, 135, 014\}$ are represented by Cluster 2, and so on. These sets of tuples form the explicit images of the respective patterns.

Since the clustering algorithm is known to the system or the analyst, the abstract form (i.e., the pattern type) of the result is known a priori. In the case of the clusters of our reference example, the result is a 2D-disk represented as follows:

Name	Cluster
Structure Schema	`disk:[Center:[X:real,Y:real],Rad:real]`
Domain	`rel:{[A1:integer, A2:integer]}`
Measure Schema	Precision: `real`
Mapping Formula	$(\texttt{rel.A1} - \texttt{disk.Center.X})^2 + (\texttt{rel.A1} - \texttt{disk.Center.}Y)^2 \leq \texttt{disk.Rad}^2$

The actual clusters for the data set `CustBranch1` are `Cluster1` and `Cluster2`. Since the clustering algorithm is applied only over the attributes `age` and `income` of

TransBranch1:{bread,butter}

id	items
193	bread, butter, milk
189	bread, butter
124	bread, beer, butter
133	bread, knife, butter

TransBranch2:{bread,butter,milk}

id	items
293	bread, butter, milk, candies
289	bread, butter, milk
224	bread, beer, butter, milk
233	bread, knife, butter, milk

Table 2.2: *Frequent Itemsets in relations* ***TransBranch1*** *and* ***TransBranch2***

CustBranch1, only these attributes appear in the Active Domain. Technically this can be achieved by using a simple view on CustBranch1.

Pid	339 (Cluster 1)
Structure	disk:[Center:[X:30,Y:30],Rad:3]
Active Domain	CustBranch1:{[age,income]}
Measure	Precision: 1
Mapping Formula	$(\texttt{CustBranch1.age} - 30)^2 + (\texttt{CustBranch1.income} - 30)^2 \leq 3^2$

Pid	340 (Cluster 2)
Structure	disk:[Center:[X:45,Y:60],Rad:2]
Active Domain	CustBranch1:{[age,income]}
Measure	Precision: 0.75
Mapping Formula	$(\texttt{CustBranch1.age} - 45)^2 + (\texttt{CustBranch1.income} - 60)^2 \leq 2^2$

In the case of the products sold, the manager wants to detect products that are often sold together, so he uses a data mining algorithm that detects frequent itemsets and among others he discovers the frequent itemsets depicted in Table 2.2. Even in this case, the pattern type is known in advance:

Name	Frequent Itemset
Structure Schema	fitems:{item: string}
Domain	rel:{[tid: integer, items: {string}]}
Measure Schema	Precision: real
Mapping Formula	fitems $\subseteq$ rel.items

2.4.4 The pattern warehouse

Having defined the data and the pattern space, we are ready to introduce the global framework within which data- and pattern-bases coexist. To this end, we formally define the relation between data and patterns and the overall context of patterns, data, and their mappings.

When a pattern is created by using a data mining algorithm, a subset of the data set used as input for the data mining algorithm is mapped to the pattern. For example, a clustering algorithm maps several data from a data set to the label of one cluster. We say that these data are *explicitly represented* by the pattern and we keep a catalog linking all these data with the pattern in the intermediate mappings. Thus, the intermediate mappings are nothing but an extensional form of the relation between the patterns and the underlying data. The *PW* offers functions to handle them, although the creation and deletion of the intermediate mappings

are not its focus. Formally, the intermediate mappings between data and patterns can be defined as follows:

Definition 2.4.9. *Let $\mathcal{P}$ be the set of all patterns in the pattern space and $\mathcal{D}$ be the set of all data appearing in the data space. An* intermediate mapping*, denoted as Φ, is a partial function: $\mathcal{P} \times \mathcal{D} \rightarrow \{true, false\}$.*

Using this definition, we can formally define the explicit representation as follows.

Definition 2.4.10. *A data item d is* explicitly represented *by a pattern p, denoted as $d \hookrightarrow p$, iff $\Phi(p, d) = true$.*

Finally, the explicit image of a pattern can be defined as follows.

Definition 2.4.11. *An* explicit image *of a pattern p, denoted as $\mathcal{I}_e(p)$, is the finite set of the data values explicitly represented by the pattern p, i.e., $\mathcal{I}_e(p) = \{x \mid x \hookrightarrow p\}$.*

To avoid any possible confusion regarding the relation between the image and the active domain of a pattern, it should be stated that the active domain concerns the data set which was used as input to the data mining algorithm, whereas the image describes the subset of the active domain that the data mining algorithm attributed to the pattern.

Remember that patterns are not linked to the data only by the intermediate mappings. Patterns are also *approximately* linked to the data through the mapping formula. The formula of each pattern is an effort to summarize the information carried by the intermediate mappings, describing as accurately as possible a criterion for selecting the data that are represented by the pattern from the original data set. In this sense, *the formula of each pattern is an approximation of the mapping* Φ. This approximate link actually carries a more generalized notion of the relation between the pattern and the data space; each pattern is mapped to a whole *region* of the data space, independently of the actual data records that exist. All points of this region are *approximately represented* by the pattern. More formally, assuming that the formula is actually a predicate called *mapping predicate*, the approximate representation is defined as follows:

Definition 2.4.12. *Let p be a pattern whose represented data lie in the domain D. An element $x \in D$ is* approximately represented *(or simply,* represented*) by pattern p, denoted as $x \rightsquigarrow p$, iff $mp(x) = true$, where mp is a predicate called the* mapping predicate *of p.*

Note that D in the above definition does not refer to the active domain of the data of a certain pattern, but rather, D refers to all the possible data records,

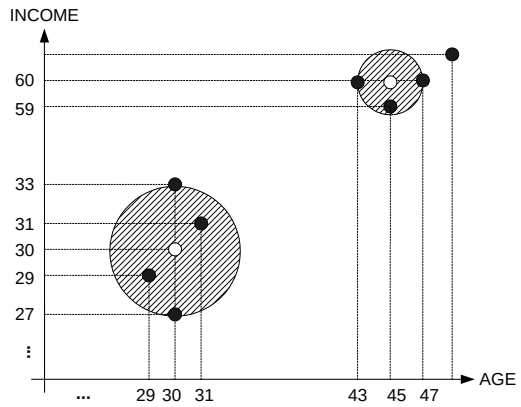


Figure 2.3: *The explicit and approximate image of Clusters 1 and 2*

that qualify for its mapping predicate. Note also that throughout the thesis, by *represent* we refer to the relationship *approximately represents*. When the semantics of Definition 2.4.10 is used we indicate the relationship as explicit representation.

Similarly to the explicit image, we can now define the *approximate image* of a pattern, or simply the image of the pattern:

Definition 2.4.13. *An* approximate image *of a pattern p, denoted by $\mathcal{I}(p)$, is the set of the data values approximately represented by the pattern p, i.e., $\mathcal{I}(p) = \{x \mid x \rightsquigarrow p\}$*

The above definitions do not specify (directly or indirectly) any relation between the approximate image $\mathcal{I}(p)$ and the explicit image $\mathcal{I}_e(p)$ of a pattern p. It is a matter of the pattern base designer to decide if some constraint (like $\mathcal{I}_e(p) \subseteq \mathcal{I}(p)$) has to be imposed. In principle, it is an implementation and administration issue whether the explicit image $\mathcal{I}_e(p)$ is saved (with the storage and maintenance cost that this implies) or all the operations are executed by using the approximate image $\mathcal{I}(p)$ (with the respective approximation error), which requires knowing only the mapping predicate. In Figure 2.3, the images of Cluster 1 and Cluster 2 (defined in Table 2.1) are depicted. The black dots are the points included in the two explicit images while the two grey areas represent the approximate images of the two clusters.

Since the relations between the pattern and the data space are so important for the aims of the PW, we need to define a logical entity that encompasses both. To this end, we introduce the notion of *pattern warehouse* which incorporates the underlying database, the pattern base, and the intermediate mappings. Although we separate patterns from data, we need the full environment in order to answer interesting queries and to support interactive user sessions navigating from the pattern to the data space and vice versa.

Definition 2.4.14. *A* pattern warehouse schema *is a pair $\langle \widehat{DB}, \widehat{PB} \rangle$, where $\widehat{PB}$ is a pattern base schema defined over the database schema $\widehat{DB}$.*

Definition 2.4.15. *A* pattern warehouse *is an instance of a pattern warehouse schema defined as a triplet:* $\langle DB, PB, \Phi \rangle$*, where* DB *is a database instance,* PB *is a pattern base instance, and* Φ *is an intermediate pattern-data mapping over* DB *and* PB*.*

2.4.5 The mapping formula

The mapping formula is the intensional description of the relation between the pattern and the data space. The mapping formula must provide a *logical criterion* enabling the identification of the data that are represented by a pattern and vice versa. In this section, we present the a formalism for defining mapping formulae and we focus on the following aspects: (a) the requirements for the formalism; (b) the language that we adopt to express it; (c) the safe incorporation of the introduced formalism in a query language; and (d) the extensibility of this formalism in terms of incorporating user-defined functions and predicates. The proposed formalism is focused on clusters and other patterns which define some sub-region of data space. Although it would be ideal to have one formalism for all types of patterns, we expect that in practice we'll need different types of formalisms for radically different pattern types.

Requirements. Given the complexity of the relation that may exist between the data and the patterns, as well as the various complicated structures of data we may face, several requirements arise for the mapping formula.

- The mapping formula must be *informative* to the user, i.e., the user must be able to intuitively understand the abstraction provided by the pattern.
- A basic requirement is the ability to handle the *complex data values and types* that appear in the structure and domain fields.
- The mapping formula must be able to express a great variety of relations between patterns and data, and it should allow simple and easy extensions.
- The information described in the mapping formula should be in a form that can be easily used by queries. This means that the mapping formula must be expressed in a generic language that facilitates reasoning methods.

From a more technical point of view, at the pattern type level, the mapping formula relates the *structure schema* with the respective *domain.* At the pattern level, the mapping formula relates the *structure*, which is an instantiation of the structure schema, with the *active domain* of the pattern, i.e., with the set value (relation in our examples) appearing in the active domain component. Intuitively, the mapping formula at the pattern type level correlates regions of the pattern

space with regions of the data space, whereas in the pattern level it correlates a single region of the data space with a specific value (the structure) in the pattern space. In the general case, the mapping formula does not have to be stored at the pattern level since it can be derived from the respective field of the pattern type and from the structure and active domain values of the pattern.

Now, we can formally define what a well-formed *mapping formula* is for a *pattern-type*.

Definition 2.4.16. *A pattern type mapping formula takes the form:*

$$mp(\overline{dv}, \overline{pv}) \tag{2.1}$$

where mp is a predicate called the mapping predicate*, $\overline{dv}$ are variable names appearing in the auxiliary relation in domain D and $\overline{pv}$ are variable names that appear in the structure schema SS.*

At instantiation time, $\overline{pv}$ is assigned values of the structure component and $\overline{dv}$ is mapped to the relation appearing in the active domain component. Similarly, we can formally define well-formed *mapping formula* for patterns.

Definition 2.4.17. *A pattern mapping formula takes the form:*

$$mp(\overline{dv}) \tag{2.2}$$

where mp is a predicate called the mapping predicate *and $\overline{dv}$ are variables appearing in the active domain component AD.*

Syntax. In the previous definitions, we chose to define the mapping formula as a predicate. This predicate is true for all the values in the domain (and of course in the active domain) of the pattern that are approximately represented by the pattern. This predicate is defined using first order logic, the logical connectives ($\neg$, $\vee$, $\wedge$ etc), and at least two constructor functions: the tuple [ ] and the set { } constructors, in order to be able to handle complex values, as they were presented in Section 2.4.1.

Extensibility. We can increase the expressive power of the mapping predicate by introducing more functions and predicates in the language we have adopted. In this context, we have *interpreted functions* and *predicates* that allow the formula to be informative to the user and offer reasoning abilities.

Safety in the use of the mapping predicate. The mapping predicate by itself is not a query, thus, there is no issue of safety. Still, we want to be able to use it in queries. For example, we would like to be able to use the mapping predicate to construct a query that would return all data from a specific dataset that can be approximately represented by the pattern. Being able to answer this query on

the data, without any further knowledge, implies that all the free variables of the predicate take values from the active domain of the pattern. This is ensured by range-restricting all variables appearing in the formula predicate to constants or to the finite datasets appearing in the active domain. This results in a *safe* query. Still, the presence of even very simple functions makes the guarantee of safety, in terms of the classical domain independence, impossible. There is a need for a notion of safety broad enough to deal with functions and predicates. We opted to define safety for the PW queries in terms of the n-depth domain independence proposed in [AB95] that allows a finite number of application of functions. In this sense, safe queries are domain-independent in the active domain is closed under n applications of functions. In other words, any query can be safely evaluated and its results can be computed and returned to the user, provided that there is an upper bound (n) to the number of times that functions can be applied in it. Similar generalized forms of domain independence have been proposed in [EMHJ93, Suc95].

2.5 Summary

The aim of this chapter was to introduce the basic foundation for pattern management, the pattern warehouse. We described how we conceive patterns as condensed forms that express some property datasets and we identified the requirements for the pattern warehouse. Using these requirements as a guide we proposed a conceptual architecture for the pattern warehouse. We showed how both patterns and data participate in an integrated environment, as distinct entities.

We proposed and described a logical model where both data and patterns are primary citizens. We defined as the basic components of the patterns, their structure, their source and their mapping formula, which connects the pattern structure with their data source. We detailed how patterns are derived from instantiating pattern types and how collections of patterns with similar semantics form pattern classes.

We described two mechanisms for expressing the relation between data and patterns. An exact one, the intermediate mappings, where a listing matching explicitly data and patterns is kept, and an approximate one, the mapping formula. The latter is an effort to approximate the relation between the pattern and the data space in some logical language, whenever possible.

The constructs and the entities we described in this chapter comprise the logical foundation for a pattern warehouse. In practice, several additions might be necessary (e.g. it is unlikely that one language will be able to express any kind of patterns in the mapping formula), but the model we presented is sufficient to support some basic functionality in terms of queries, that is not currently supported by any other

system. Relations between patterns and query operators is exactly our focus in the next Chapter.

Chapter 3

Querying the Pattern Warehouse

In Chapter 2 we presented the conceptual architecture of the Pattern Warehouse, and we proposed a logical model for its organization. The aim of this Chapter is to show how we can exploit the pattern warehouse to explore huge collections of data and patterns.

To unlock the potentiality of the pattern warehouse, it is of paramount importance to identify the possible relations between the patterns and to discover how to built upon existing patterns, in order to gain new knowledge about the underlying dataset. To this end, we define a set of pattern relations and we prove that it is sound and complete. Moreover, we propose operators for composing patterns from existing ones. We complement our efforts by presenting a series of operators that take advantage of these relations for query the pattern warehouse.

3.1 Querying the pattern warehouse

Based on the definitions of Chapter 2, we can now proceed in investigating how we can exploit the information stored in it. The pattern warehouse is a larger environment than the pattern base, as it incorporates patterns, data, and intermediate mappings. Therefore, the user can potentially pose queries over data, patterns, and their intermediate relationships. In this chapter, we will address the following fundamental query operations over a pattern warehouse:

- What are the possible relationships between two individual patterns?
- Given two patterns, can we combine them to produce a new one, or can we compare them to find out how similar they are?
- Given a set of patterns, i.e., a pattern class, or possibly, a set of classes, what kind of information can we extract by querying them?

- How can we possibly navigate back and forth between the pattern base and the raw data? What kind of interesting queries can we pose over the pattern-data interrelationships?

Our basic contribution in this chapter is the definition of a sound and complete set of binary pattern relations. We define the semantics of each pattern relation in the data space, and how each relation between two patterns is reflected in the properties of the underlying data. Moreover, we prove that the proposed set of relations is sound and complete. The usefulness of these relations is demonstrated by the constructors and the query operators we present from our joint work in [TVS+04a, TVS+]. The proposed constructors provide the union, difference and similarity of two patterns. Moreover, we provide operators similar to the relational ones for the management of pattern classes and investigate their particularities when applied to collections of patterns. Finally, we provide three operators for navigating between the pattern and the data space.

3.1.1 Predicates on patterns

In this section, we investigate relations between patterns, using both their extensional and their intensional representation, and propose a complete set of predicates that characterize all possible relations between two patterns. To this end, we treat patterns as point sets in the data space. In fact, as defined in Section 2.4.4, patterns actually correspond to two possibly different point-sets in the data space: (a) a finite set comprising all data which are explicitly represented by the pattern, i.e., there exist intermediate mappings linking them to the pattern, and (b) a possibly infinite set described by the formula predicate, i.e., all points of this set are approximately represented by the pattern. Naturally, the former refers to the explicit image of the pattern and the latter to the approximate image of the pattern.

In all cases, we will define our binary pattern operators in two flavors:

- *Explicit relationships.* Explicit relationships are defined in terms of the specific data that patterns represent, i.e., with respect to their explicit image, as obtained through intermediate mappings.

- *Approximate relationships.* Approximate relationships are defined in terms of the approximately represented values of the data space, i.e., with respect to the approximate images of the involved patterns.

We introduce three fundamental pattern operators, with semantics similar (but not identical) to the respective set-theoretic ones. Two given patterns p_A and p_B can satisfy exactly one of the following:

- p_A *subsumes* p_B (denoted as $p_A \succ p_B$), meaning that the image of p_A is a superset of the one of p_B,
- p_A and p_B are *disjoint* (denoted as $p_A \pitchfork p_B$), meaning that their images in the data space have no common points, or
- p_A and p_B *intersect* (denoted as $p_A \cap p_B$), meaning that their images in the data space have at least one common and one not common point.

Before specifying the above operators, we define the notion of *data compatible patterns* that clarifies when it is meaningful to compare two patterns with respect to the underlying data.

Definition 3.1.1. *Two pattern types are* data compatible *if the data types described in their data schemata describe the same domain. Two patterns are data compatible if they are instances of pattern types that are data compatible.*

In the sequel, first we introduce our pattern operators and then we prove that they are complete and minimal.

3.1.1.1 Intersecting patterns

The first test that we can pose on two patterns concerns the possibility of these two patterns having common data in the data space. The notion of pattern intersection is slightly different from the well-known, set-theoretic intersection. In our case, we require that (a) at least a common data item exists in the explicit images of both patterns involved in the comparison (as usual), (b) there exists at least one data item that corresponds to the first pattern, but does not correspond to the second one (i.e., their difference is not empty) and (c) the previous relationship also holds the other way. Our deviation from the traditional definition practically identifies patterns that are neither identical, nor disjoint. This is necessary for providing a minimal and complete set of comparison operators for two patterns and it will be made evident in the sequel (Theorem 3.1.1).

As already mentioned, we provide two variants for every binary operator, so we define an explicit and an approximate variant of intersection.

Definition 3.1.2. *Two data compatible patterns p_A and p_B* explicitly intersect*, denoted as $p_A \cap_e p_B$, iff $(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\exists y, z)(y \hookrightarrow p_A \wedge z \hookrightarrow p_A \wedge \neg(y \hookrightarrow p_B) \wedge \neg(z \hookrightarrow p_B))$.*

Definition 3.1.3. *Two data compatible patterns p_A and p_B* approximately intersect*, denoted as $p_A \cap p_B$, iff $(\exists x)(x \rightsquigarrow p_A \wedge x \rightsquigarrow p_B) \wedge (\exists y, z)(y \rightsquigarrow p_A \wedge z \rightsquigarrow p_B \wedge \neg(y \rightsquigarrow p_B) \wedge \neg(z \rightsquigarrow p_A))$.*

3.1.1.2 Disjoint patterns

As usual, two patterns are disjoint if they do not share any common item in the data space. With respect to their formulae, two patterns intersect when they are defined over the same domain and the conjunction of their formula predicates is satisfied. If the formula cannot be satisfied in the domain D then they are disjoint. The *disjoint* relationship is given in the following two definitions (for the explicit and the approximate variant).

Definition 3.1.4. *Two data compatible patterns p_A and p_B are* explicitly disjoint*, denoted as $p_A \pitchfork_e p_B$, iff $(\nexists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B)$.*

Definition 3.1.5. *Two data compatible patterns p_A and p_B are* approximately disjoint *(or simply,* disjoint*), denoted as $p_A \pitchfork p_B$, iff $(\nexists x)(x \rightsquigarrow p_A \wedge x \rightsquigarrow p_B)$.*

3.1.1.3 Pattern equivalence

A pattern p_A is equivalent to another pattern p_B when all data records represented by p_B are also represented by p_A and vice versa. The explicit and the approximate variant of the equivalence relationship is defined as follows.

Definition 3.1.6. *Let p_A and p_B be two data compatible patterns. Pattern p_A and p_B are* explicitly equivalent*, denoted as $p_A \equiv_e p_B$, iff $(\forall x)\big((x \hookrightarrow p_B) \Leftrightarrow (x \hookrightarrow p_A)\big)$.*

Definition 3.1.7. *Let p_A and p_B be two data compatible patterns. Pattern p_A and p_B are* approximately equivalent *(or simply,* equivalent*), denoted as $p_A \equiv p_B$, iff $(\forall x)\big((x \rightsquigarrow p_B) \Leftrightarrow (x \rightsquigarrow p_A)\big)$.*

We also use $p_A \not\equiv p_B$ (respectively $p_A \not\equiv_e p_B$) to denote that p_A and p_B are *not* equivalent (respectively explicit equivalent).

3.1.1.4 Pattern subsumption

A pattern p_A subsumes another pattern p_B when (a) p_A and p_B are not equivalent and (b) all data records represented by p_B are also represented by p_A. For example, if the formula predicate of p_A is $x > 5$ and the formula predicate of p_B is $6 < x < 8$, and x is defined over the same domain D, then, we can say that p_A approximately subsumes p_B. The explicit and the approximate variant of the subsumes relationship is defined as follows.

Definition 3.1.8. *Let p_A and p_B be two data compatible patterns. Pattern p_A* explicitly subsumes *p_B, denoted as $p_A \succ_e p_B$, iff (a) $p_A \not\equiv_e p_B$ and (b) $(\forall x)\big((x \hookrightarrow p_B) \Rightarrow (x \hookrightarrow p_A)\big)$.*

Definition 3.1.9. *Let p_A and p_B be two data compatible patterns. Pattern p_A* approximately subsumes *(or simply,* subsumes*) p_B, denoted as $p_A \succ p_B$, iff (a) $p_A \not\equiv p_B$ and (b) $(\forall x)\big((x \rightsquigarrow p_B) \Rightarrow (x \rightsquigarrow p_A)\big)$.*

3.1.1.5 A complete set of binary pattern relations

Given two data compatible patterns p_A and p_B, we can easily prove that, with respect to their images, they have exactly one of the previously introduced relationships: (a) they are disjoint (i.e., they do not have any common data item in the data space), (b) they are equivalent, (c) one subsumes the other (i.e., all data of one pattern belong to the corresponding data of the other), or (d) they have some, but clearly not all data in common. This property holds for both the explicit and the approximate images of data. In other words, the introduced set of relationships is *complete*. We can easily prove that the introduced set is *minimal*. These properties are formalized in the following theorems.

Theorem 3.1.1. *Every two data compatible patterns p_A and p_B satisfy exactly one of the relations $\{p_A \cap_e p_B$, $p_A \pitchfork_e p_B$, $p_A \equiv_e p_B$, $p_A \succ_e p_B$, $p_B \succ_e p_A\}$ with respect to their explicit images.*

Proof. (Sketch) It is easy to see that the previous relations are mutually exclusive, since we have paid attention that equivalent patterns do not satisfy the intersect of the subsume relation. Given that they are mutually exclusive it is also trivial to see that they are minimal: we only need to find one example for each different relation.

To prove that the above set of relations is complete, we will exploit the tautology $\alpha \vee (\neg\alpha) \Leftrightarrow true$. Specifically, for any data compatible patterns p_A and p_B the following hold:

1. $(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \vee \neg(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \Leftrightarrow$

2. $(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \vee \neg(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \Leftrightarrow$

3. $(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_A)) \vee p_A \pitchfork_e p_B \Leftrightarrow$

4. $[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge$
   $[(\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \vee \neg(\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B))] \wedge$
   $[(\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A)) \vee \neg(\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A))]]$
   $\vee p_A \pitchfork_e p_B \Leftrightarrow$

5. $[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge$
   $[(\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \vee (\exists x)(x \in \mathcal{I}_e(p_A) \wedge \neg(x \in \mathcal{I}_e(p_B)))] \wedge$
   $[(\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A)) \vee (\exists x)(x \in \mathcal{I}_e(p_B) \wedge \neg(x \in \mathcal{I}_e(p_A)))]]$
   $\vee p_A \pitchfork_e p_B \Leftrightarrow$

6. $[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge$
$[(\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \wedge (\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A)) \vee$
$(\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \wedge (\exists x)(x \in \mathcal{I}_e(p_B) \wedge \neg(x \in \mathcal{I}_e(p_A))) \vee$
$(\exists x)(x \in \mathcal{I}_e(p_A) \wedge \neg(x \in \mathcal{I}_e(p_B))) \wedge (\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A)) \vee$
$(\exists x)(x \in \mathcal{I}_e(p_A) \wedge \neg(x \in \mathcal{I}_e(p_B))) \wedge (\exists x)(x \in \mathcal{I}_e(p_B) \wedge \neg(x \in \mathcal{I}_e(p_A)))]]$
$\vee p_A \pitchfork_e p_B \Leftrightarrow$

7. $[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge (\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \wedge (\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A))] \vee$
$[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge (\forall x)(x \in \mathcal{I}_e(p_A) \Rightarrow x \in \mathcal{I}_e(p_B)) \wedge (\exists x)(x \in \mathcal{I}_e(p_B) \wedge \neg(x \in \mathcal{I}_e(p_A)))] \vee$
$[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge (\exists x)(x \in \mathcal{I}_e(p_A) \wedge \neg(x \in \mathcal{I}_e(p_B))) \wedge (\forall x)(x \in \mathcal{I}_e(p_B) \Rightarrow x \in \mathcal{I}_e(p_A))] \vee$
$[(\exists x)(x \in \mathcal{I}_e(p_A) \wedge x \in \mathcal{I}_e(p_B)) \wedge (\exists x)(x \in \mathcal{I}_e(p_A) \wedge \neg(x \in \mathcal{I}_e(p_B)) \wedge (\exists x)(x \in \mathcal{I}_e(p_B) \wedge \neg(x \in \mathcal{I}_e(p_A)))] \vee$
$p_A \pitchfork_e p_B \Leftrightarrow$

8. $[(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\forall x)(x \hookrightarrow p_A \Rightarrow x \hookrightarrow p_B) \wedge (\forall x)(x \hookrightarrow p_B \Rightarrow x \hookrightarrow p_A)] \vee$
$[(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\forall x)(x \hookrightarrow p_A \Rightarrow x \hookrightarrow p_B) \wedge (\exists x)(x \hookrightarrow p_B \wedge \neg(x \hookrightarrow p_A))] \vee$
$[(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\exists x)(x \hookrightarrow p_A \wedge \neg(x \hookrightarrow p_B)) \wedge (\forall x)(x \hookrightarrow p_B \Rightarrow x \hookrightarrow p_A)] \vee$
$[(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\exists x)(x \hookrightarrow p_A \wedge \neg(x \hookrightarrow p_B) \wedge (\exists x)(x \hookrightarrow p_B \wedge \neg(x \hookrightarrow p_A))] \vee$
$p_A \pitchfork_e p_B \Leftrightarrow$

9. $[(\exists x)(x \hookrightarrow p_A \wedge x \hookrightarrow p_B) \wedge (\forall x)(x \hookrightarrow p_A \Rightarrow x \hookrightarrow p_B) \wedge (\forall x)(x \hookrightarrow p_B \Rightarrow x \hookrightarrow p_A)] \vee$
$p_A \succ_e p_B \vee$
$p_B \succ_e p_A \vee$
$p_A \cap_e p_B \vee$
$p_A \pitchfork_e p_B \Leftrightarrow$

10. $p_A \equiv_e p_B \vee$
$p_A \succ_e p_B \vee$
$p_B \succ_e p_A \vee$
$p_A \cap_e p_B \vee$
$p_A \pitchfork_e p_B$

Based on the completeness and minimality of the aforementioned set of relations, we can therefore conclude that they are sufficient to characterize the relationship of any two patterns. □

Theorem 3.1.2. *Every two data compatible patterns p_A and p_B satisfy exactly one of the relations $\{p_A \cap p_B,\ p_A \pitchfork p_B,\ p_A \equiv p_B,\ p_A \succ p_B,\ p_B \succ p_A\}$ with respect to their approximate images.*

Proof. The proof follows the same steps with the proof of Theorem 3.1.1 using the approximate images of the patterns instead of the explicit ones. □

3.1.1.6 Equality

So far, we have referred to pattern equivalence with respect to the images of the involved patterns. Naturally, except for the predicates that are defined with respect to the images of a pattern, we can define identity or equality in a broader sense. Specifically, we give two different definitions of equality:

- *Identity* (=). Two patterns p_1 and p_2 are identical if they have the same PID, i.e., $p_1.PID = p_2.PID$.

- *Shallow equality* ($=^s$). Two patterns p_1 and p_2 are shallow equal if their corresponding components (except for the PID component) are equal, i.e., $p_1.S = p_2.S$, $p_1.AD = p_2.AD$, $p_1.M = p_2.M$, and $p_1.MF = p_2.MF$. Note that, to check the equality for each component pair, the basic equality operator for the specific component type is used. It is easy to verify that $p_1 =^s p_2 \Rightarrow p_1 \equiv p_2$.

3.1.1.7 The similarity operator ξ

Similarity between patterns is not an obvious notion. To the best of our knowledge there has been little work on the similarity of patterns [GRGL99, BCN+04], and none of it is dealing with similarity of patterns of different types. As in [GRGL99] our notion of similarity is defined with respect to the data. Actually, as we did in the previous subsection, we define here two different versions of the similarity operator: (a) the *explicit similarity operator* ξ_e defined on $\mathcal{I}_e$, and (b) the *approximate similarity operator* ξ defined on $\mathcal{I}$.

The explicit similarity operator. The explicit similarity operator ξ_e is defined over the similarity of the represented datasets $\mathcal{I}_e$. A straightforward approach to defining this similarity would be to take into account how many common values exist in the explicit images of the compared patterns. One of the possible measures for such a similarity definition is the Jaccard coefficient:

Definition 3.1.10. *Let p_A and p_B be two patterns. The* explicit similarity *of p_A and p_B, denoted as $\xi_e(p_A, p_B)$, is defined by the expression* $\xi_e(p_A, p_B) = \frac{COUNT(\mathcal{I}_e(p_A) \bigcap \mathcal{I}_e(p_B))}{COUNT(\mathcal{I}_e(p_A) \bigcup \mathcal{I}_e(p_B))}$

Naturally, other measures apply, mainly from the field of Information Retrieval, involving the simple matching, cosine, or Dice's coefficients [Dun02].

The approximate similarity operator. Definition 3.1.10 does not always reflect the similarity of the knowledge carried by the patterns. For example, in the case depicted in Figure 3.1 the two clusters carry approximately the same information: the fact that there is a concentration of employees around the age 30-31 with the same salary of 30k. Whereas this is visually obvious, it is not directly

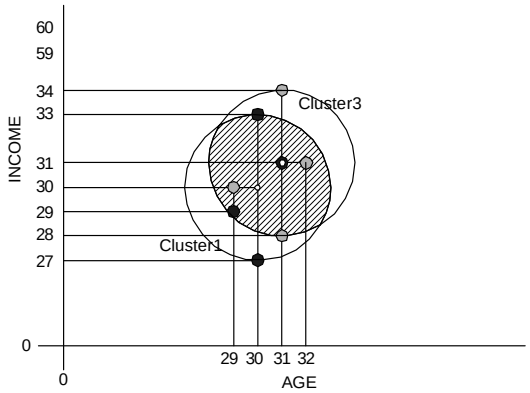


Figure 3.1: *Pattern similarity*

reflected in the relations between the two different data sets that are represented by the clusters, since they do not share even one common value. What we are actually comparing in this case are the whole areas in the data space that are defined by the pattern formulae and not the areas of the actual data. A measure for similarity that captures this notion is defined similarly to the explicit similarity.

Definition 3.1.11. *Let p_A and p_B be two data compatible patterns. The* similarity *of p_A and p_B, denoted as ξ, is defined by the expression $\xi(p_A, p_B) = \frac{V(\mathcal{I}(p_A) \bigcap \mathcal{I}(p_B))}{V(\mathcal{I}(p_A) \bigcup \mathcal{I}(p_A))}$, where V is a function that gives the volume of the region in the n-dimensional space of the image.*

Intuitively, the similarity of two patterns is defined as the ratio of the size of their intersection divided by the size of their union. Thus, two disjoint patterns have $\xi = 0$ and two equal patterns have $\xi = 1$. Whereas in the case of the disk clusters depicted in Figure 3.1, we could define the distance between their centers as a similarity measure, defining similarity over the size of their intersection and their union has an important property: *similarity is independent of the structure type of a pattern*. Thus, we can define the similarity between patterns that have different structures, i.e., they define regions with different shapes in the data space. Observe that since the data space in principle involves n attributes, the regions are defined over a n-dimensional space.

If the union or the intersection of the images of the patterns are not constrained in one or more dimensions, then we cannot define the approximate similarity operator. In this case, if spatial reasoning algorithms cannot be employed, we can still compute the similarity between the patterns by using the explicit similarity operator.

3.1.2 Pattern constructors

Having introduced all possible relationships among two patterns, we can now define pattern constructors resulting from such relations. Therefore, we introduce the

intersection and union of two patterns. Then, we discuss issues of pattern similarity.

3.1.2.1 The *intersection* and *union* constructors

Constructing new patterns from new data sets is primarily a task for data mining methods; nevertheless, we can construct new patterns from the existing ones, in the PW, too.

Definition 3.1.12. *Let* $p_A = (pid_A, s_A, ad_A, m_A, mp_A)$ *and* $p_B = (pid_B, s_B, ad_B, m_B, mp_B)$ *be two data compatible patterns. The* intersection *of patterns* p_A *and* p_B*, denoted as* $p_A \bigcap p_B$*, is defined as*

$$p_A \bigcap p_B = \big(newpid(), \{s_A, s_B\}, ad, f(m_A, m_B), mp_A \wedge mp_B\big),$$

where $ad = \{x \mid (x \in ad_A \vee x \in ad_B)\}$ *and* f *some measure composition function. Furthermore,* $\mathcal{I}_e(p_A \bigcap p_B) = \{x \mid (x \in \mathcal{I}_e(p_A) \vee x \in \mathcal{I}_e(p_B)) \wedge (x \rightsquigarrow p_A \wedge x \rightsquigarrow p_B)\}$.

Practically, the pattern that is created by the intersection of two other patterns p_A and p_B consists of:

- A new *pid* created by some system function.
- A new structure $\{s_A, s_B\}$, which is a set having as members the values s_A and s_B that appear in the structure field of p_A and p_B respectively.
- A new relation $ad_{A \bigcup B}$ in the data field that contains all the data that were specified in the *data* fields of the input patterns. Note that since p_A and p_B are data compatible, the relations appearing in their data fields have compatible type, so the union of their values can appear under one of the initial relation schemas.
- A new measure $f(m_A, m_B)$, computed by employing a measure composition function f. The measure computation mechanism is currently a direction for future work.
- A new formula that is the conjunction of the two formula predicates of the intersecting patterns. Some attention should be paid in mapping the variables from the initial relations to the resulting one. We might be able to further simplify this formula using techniques from constraint theory. For example, if the two formula predicates of p_A and p_B are $mp_A = 5 < x < 7$ and $mp_B = 4 < x < 6$, then we can reduce $mp_A \wedge mp_B$ to $5 < x < 6$.

Note that the data mappings are created *based on the intensional description of the pattern*. That is, from the explicit images of the intersecting patterns, only

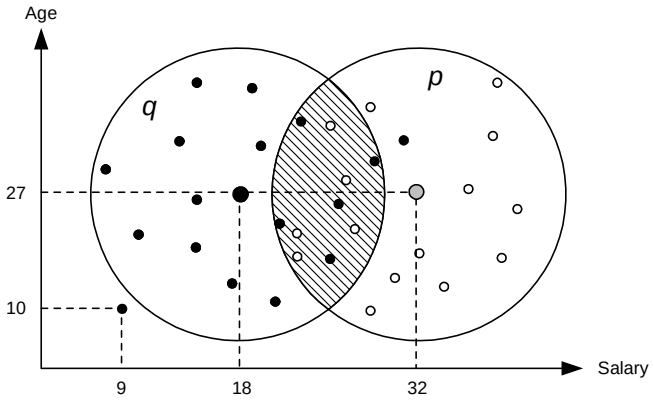


Figure 3.2: *The intersection operator. Solid points represent data items in the image of p_A and hollow points, represent data items in the image of p_B. The explicit images do not have any common point.*

those data satisfying the formula predicate of the new pattern, i.e., those that are approximately represented by the new pattern, are actually mapped to the intersection pattern. Figure 3.2 graphically illustrates this remark. The new pattern describes the shadowed area, thus, we would like to map onto it all the data inside the shadowed area. Note that data explicitly represented by p_A or p_B are in the intersection, but they are not *common* (i.e., none of them belongs to both datasets). Thus, if we mapped the data that are in the intersection of $\mathcal{I}_e^{p_A}$ and $\mathcal{I}_e^{p_B}$, instead of those that fall in the intersection of $\mathcal{I}^{p_A}$ and $\mathcal{I}^{p_B}$, we would end up with no data mapped to the new pattern. Therefore, we assume that pattern formulae are broader generalizations of explicit images (i.e., no point belonging to the explicit image of a pattern is lost from the pattern formula) and we compute the new explicit image based on the original approximate images. From our point of view, following the explicit intermediate mappings offers a rather narrow approach that we chose to completely avoid in this chapter.

Definition 3.1.13. *Let $p_A = (pid_A, s_A, ad_A, m_A, mp_A)$ and $p_B = (pid_B, s_B, ad_B, m_B, mp_B)$ be two data compatible patterns. The* union *of patterns p_A and p_B, denoted as $p_A \bigcup p_B$, is defined as*

$$p_A \bigcup p_B = \big(newpid(), \{s_A, s_B\}, ad, f(m_A, m_B), mp_A \vee mp_B\big),$$

where $ad = \{x \mid x \in ad_A \vee x \in ad_B\}$ and f some measure composition function. Furthermore $\mathcal{I}_e(p_A \bigcup p_B) = \{x \mid (x \in \mathcal{I}_e(p_A) \vee x \in \mathcal{I}_e(p_B))\}$.

Practically, the new pattern resulting from the union operator is created as follows:

- First, the pattern gets a new *pid* from some system function.

- The *structure* of the pattern is a set composed of the values that appeared in the *structure* component of the initial patterns, exactly as in the case of *intersection*.

- The pattern *data* is the union of the *data* of the initial patterns.

- The new *measure* is computed, again, by employing a measure composition function, f.

- The new *formula* is the disjunction of the formulae of the initial patterns. Again reduction techniques can be applied as for the case of the *intersection*.

Lemma 3.1.1. *Let p_A and p_B be two patterns. Then $(x \hookrightarrow p_A \vee x \hookrightarrow p_B) \Rightarrow (x \hookrightarrow p_{A \bigcup B})$.*

Proof. Obvious. □

Note that based on the above result, it is not required that $\mathcal{I}_e(p_A \bigcup p_B) = \mathcal{I}_e(p_A) \bigcup \mathcal{I}_e(p_B)$ as one might expect. This happens only if there has been imposed a constraint which requires that $\mathcal{I}_e \subseteq \mathcal{I}$.

3.2 Query Operators

The predicates we defined for patterns are a significant tool in our effort to provide methods that allow investigating large datasets. This is demonstrated by the operators we defined in cooperation with Barbara Catania and Anna Maddalenna in [TVS+04a, TVS+]. In this Section, we present pattern constructors, query operators that manipulate pattern classes and query operators that facilitate the navigation from the pattern to the data space in detail.

3.2.1 Operators defined over pattern classes

In the following, we introduce several operators defined over pattern classes. Some of them, like set-based operators, renaming and selection are quite close to their relational counterparts; nevertheless, some others, like join and projection have significant differences. Examples of the proposed operators will be provided in Section 3.2.3.

Set-based operators. Since classes are sets of patterns, usual set operators such as union, difference and intersection are defined for pairs of classes with compatible pattern types. The criterion for equality/identity of the patterns can be any of the ones mentioned in the previous subsection.

Renaming. Similarly to the relational context, we consider a renaming operator μ that takes a class and a renaming function and changes the names of the pattern component attributes according to the specified function. Each attribute is denoted by a path expression.

Measure projection. The measure projection operator reduces the measures of the input patterns by projecting out some components. Note that this operation does not modify the pattern mapping formula, since measures do not appear in it.

Let c be a class of pattern type pt. Let lm a list of attributes appearing in $pt.Measure$. Then, the measure projection operator is defined as follows:

$\pi^m_{lm}(c) = \{(newid(), s, d, \pi_{lm}(m), f)|\ (\exists p \in c)(p = (pid, s, d, m, f))\}$

In the previous definition, $\pi_{lm}(m)$ is the usual relational projection of the measure component.

Reconstruction. The reconstruction operator allows us to manipulate the pattern structure. This operation is useful for example for projecting out some components or for nesting or unnesting data. Since the type of the pattern structure is any type admitted by the type system, we define reconstruction as a query over the pattern structure, defined by using traditional complex value algebraic operators [AB95].

More formally, let c be a class of pattern type pt. Let $I, I_1, I_2, ...$ be sets of values of type $\tau, \tau_1, \tau_2, ...$ Let q be a query constructed by using the following subset of the nested relational algebra operators plus usual set-based operators [AB95].

Projection Let I be a set of type $\tau = [B_1 : \tau_1, ..., B_k : \tau_k]$. Then, $\pi_{B_1,...,B_l}(I)$ is a set of type $[B_1 : \tau_1, ..., B_l : \tau_l]$ such that $\pi_{B_1,...,B_l}(I) = \{[B_1 : v_1, ..., B_l : v_l]|\exists v_{l+1}, ..., v_k$ such that $[B_1 : v_1, ..., B_l : v_l, B_{l+1} : v_{l+1}, ..., B_k : v_k] \in I\}$

Constructors We consider two operations, *tup_create* and *set_create*.

- If $A_1, ..., A_m$ are distinct attributes, $tup\_create_{A_1,...,A_m}(I_1, ..., I_m)$ is of type $[A_1 : \tau_1, ..., A_m : \tau_m]$ and $tup\_create_{A_1,...,A_m}(I_1, ..., I_m) = \{[A_1 : v_1, ..., A_m : v_m]|\forall v_i \in I_i, i = 1, ..., m\}$.
- $set\_create(I)$ is a set of type $\{\tau\}$ and $set\_create(I) = \{I\}$.

Destructors We consider two operations, *tup_destroy* and *set_destroy*.

- If I is of type $\{\tau\}$, and $\tau = \{\tau_1\}$, then $set\_destroy(I)$ is a set of type τ_1 and $set\_destroy(I) = \{w|\exists v \in I, w \in v\}$.
- If I is of type $[A : \tau_1]$, $tup\_destroy(I)$ is a set of type τ_1 and $tup\_destroy(I) = \{v|[A : v] \in I\}$.

Nesting and unnesting Let I_1 be a set of type t_1 such that $t_1 = [A_1 : \tau_1, ..., A_k : \tau_k, B : \{[A_{k+1} : \tau_{k+1}, ..., A_n : \tau_n]\}]$. Let I_2 be a set of type t_2 such that $t_2 = [A_1 : \tau_1, ..., A_k : \tau_k, A_{k+1} : \tau_{k+1}, ..., A_n : \tau_n]$. Then,

- $unnest_B(I_1)$ is a set of type t_2 and $unnest_B(I_1) = \{[A_1 : v_1, ..., A_n : v_n] | \exists y[A_1 : v_1, ..., A_k : v_k, B : y] \in I_1 \wedge [A_{k+1} : v_{k+1}, ..., A_n : v_n] \in y\}$;
- $nest_{B=(A_{k+1},...,A_n)}(I_2)$ is a set of type t_1 and $nest_{B=(A_{k+1},...,A_n)}(I_2) = \{[A_1 : v_1, ..., A_k : v_k, B : \{[A_{k+1} : v_{k+1}, ..., A_n : v_n]\}] | [A_1 : v_1, ..., A_n : v_n] \in I_2\}$.

Then, the reconstruction operator is defined as follows:

$\rho_q(c) = \{(pid, q(s), d, m, f) | p = (pid, s, d, m, f) \in c\}$.

Note that the formula of the resulting patterns does not have to be updated since the only variables appearing in it come from the data source, which is not changed by the operation.

Selection. The selection operator allows one to select from a given class the patterns satisfying a certain condition, involving any pattern component. Conditions are constructed by combining through logical connectives ($\wedge$, $\vee$, $\neg$) atomic formulae corresponding to: (i) predicates over patterns presented in Section 3.1.1; (ii) similarity operators introduced in Section 3.1.1.7; (iii) predicates over pattern components, depending on their data type. Predicates of type (i) and (ii) must be used with respect to a constant pattern. Predicates of type (iii) are applied to pattern components s, ad, and m which, according to the proposed model, may have a complex value type. Therefore, predicates for complex values - e.g., equality ($=$), membership ($\in$), and set-containment ($\subseteq$) - can be used according to the proper type restrictions.

Let c be a class of pattern type pt. Let pr be a predicate. Then, the selection operator is defined as follows:

$\sigma_{pr}(c) = \{p | p \in c \wedge pr(p) = true\}$

Note that, since predicates over pattern components can only be applied to tuples, in order to select patterns with respect to inner components of the structure, a reconstruction operator may have to be applied before applying the selection operator.

Join. The join operation provides a way to combine patterns belonging to two different classes according to a join predicate and a composition function specified by the user.

Let c_1 and c_2 be two classes over two pattern types pt_1 and pt_2. A join predicate F is any predicate defined over a component of patterns in c_1 and a component of patterns in c_2. A composition function c for pattern types pt_1 and pt_2 is a 4-tuple of functions $c = (c_{StructureSchema}, c_{Domain}, c_{MeasureSchema}, c_{Formula})$, one for each pattern component. For example, function $c_{StructureSchema}$ takes as input two structure values of the correct type and returns a new structure value, for a possible new pattern type, generated by the join. Functions for the other pattern components are similarly defined. Given two patterns $p_1 = (pid_1, s_1, d_1, m_1, f_1) \in c_1$ and $p_2 =$

$(pid_2, s_2, d_2, m_2, f_2) \in c_2$, $c(p_1, p_2)$ is defined as the pattern p with the following components:
$Structure : c_{StructureSchema}(s_1, s_2)$
$Domain : c_{Domain}(d_1, d_2)$
$Measure : c_{MeasureSchema}(m_1, m_2)$
$Formula : c_{formula}(f_1, f_2)$.

The join of c_1 and c_2 with respect to the join predicate F and the composition function c, denoted by $c_1 \bowtie_{F,c} c_2$, is now defined as follows:

$c_1 \bowtie_{F,c} c_2 = \{c(p_1, p_2) | p_1 \in c_1 \wedge p_2 \in c_2 \wedge F(p_1, p_2) = true\}$.

The general definition of join can be customized by considering specific composition operators. For example, by considering the union and intersection operators presented in Section 3.1.2.1, we can define the union join ($\bowtie^{\cup}$) and the intersection join ($\bowtie^{\cap}$) as follows:

$c_1 \bowtie_F^{\cup} c_2 = \{p_1 \cup p_2 | p_1 \in c_1 \wedge p_2 \in c_2 \wedge F(p_1, p_2) = true\}$.
$c_1 \bowtie_F^{\cap} c_2 = \{p_1 \cap p_2 | p_1 \in c_1 \wedge p_2 \in c_2 \wedge F(p_1, p_2) = true\}$.

3.2.2 Cross-over operators

Finally, we provide three operators for the navigation from the pattern to the data space and back. The operators are applicable to classes of patterns or data. Nevertheless, having defined the selection operator, it is straightforward to apply them to individual patterns, too.

Drill-Through. The drill-through operator allows one to navigate from the pattern layer to the raw data layer. Such operator, thus, takes as input a pattern class and returns a raw data set. More formally, let c be a class of pattern type pt which contains patterns $p_1, \ldots, p_n$. Then, the drill-through operator is denoted by $\gamma(c)$ and it is formally defined as follows: $\gamma(c) = \{d | \exists p, p \in c \ \wedge \ d \hookrightarrow p\}$ which is equivalent to: $\gamma(c) = \mathcal{I}_{\urcorner}(\sqrt{}^{\infty}) \bigcup \cdots \bigcup \mathcal{I}_{\urcorner}(\sqrt{}^{\backslash})$.

Data covering. Given a pattern p and a dataset D, sometimes it is important to determine whether p represents D or not. In other words, we wish to determine the subset S of D represented by p. Extended to a class c, this corresponds to determining the subsets S of D containing tuples represented by at least one pattern in c. To determine S, we use the mapping predicate as a query on the dataset. Let c be a class and D a dataset with schema $(a_1, ..., a_n)$, compatible with the source schema of p. The data covering operator, denoted by $\theta_d(c, D)$, returns a new dataset corresponding to all tuples in D represented by a pattern in c. More formally $\theta_d(c, D) = \{t \mid t \in D \wedge p \in c \wedge t \rightsquigarrow p\}$.

Note that, since the drill-through operator uses the intermediate mappings and the data covering operator uses the mapping formula, the covering $\theta_d(c, D)$ of the

data set $D = \gamma(c)$ returned by the drill-through operator might not be the same. This is due to the approximating nature of the pattern formula.

Pattern covering. Sometimes it can be useful to have an operator that, given a class of patterns and a dataset, returns all patterns in the class representing that dataset (a sort of inverse data covering operation). Let c be a pattern class and D a dataset with schema $(a_1, ..., a_n)$, compatible with the source schema of the c pattern type. The pattern covering operator, denoted by $\theta_p(c, D)$, returns a set of patterns corresponding to all patterns in c representing D. More formally:

$$\theta_p(c, D) = \{p \mid p \in c \wedge \forall t \in D \; t \rightsquigarrow p\}.$$

Note that:

$\theta_p(c, D) = \{p \mid p \in c \wedge \theta_d(\{p\}, D) = D\}$

For some examples concerning the application of the proposed cross-over operators, we refer the reader to Section 3.2.3.

3.2.3 Motivating example revisited

Coming back to our motivating example (Section 5.1), we can translate the description of user operations into simple queries or operations over the pattern base. A data mining algorithm is executed over the underlying data and the results are stored in the pattern base. Assume that an algorithm generating frequent itemsets is employed and the results are stored in two classes named $\texttt{FIBranch}_1$ and $\texttt{FIBranch}_2$.

Which are the data represented by a pattern? The user selects an interesting pattern from the class $\texttt{FIBranch}_1$ and decides to find out which are the underlying data that relate to it. To this end, the drill through operator can be used. Thus, assuming that the user picks the pattern with $PID = 193$, then the operation performed is:

$$\gamma(\sigma_{PID=193}(\texttt{FIBranch}_1))$$

The result is stored in a transient data set $\texttt{Result}_1^d$.

Which are the patterns representing a data item? Out of a vast number of data that the user receives as an answer, he picks a random record and decides to see which other patterns relate to it (so that he can relate the originating pattern with these ones, if this is possible). To this end, the pattern covering operator can be used. Thus, assuming that the user picks a tuple t_1 with $RID = 733$, then the operation is:

$$\theta_p(\texttt{FIBranch}_1, t_1)$$

The result is stored in a transient pattern class $\texttt{Result}_1^p$.

Is a pattern suitable for representing a data set? The user artificially generates a new dataset and checks what kinds of patterns it supports. Then, on the basis of the result, he picks a certain pattern and tries to determine which set of data corresponds to this pattern. The user creates the data set $\texttt{Cust}_3 = \sigma_{AGE>40}(\texttt{CustBranch}_1 \cup \texttt{CustBranch}_2)$ and uses again the pattern covering operator:

$$\theta_d(\texttt{FIBranch}_1 \cup \texttt{FIBranch}_2, \texttt{Cust}_3)$$

Again, the user can navigate back and forth between data and patterns through the aforementioned operators.

Could we derive new patterns from knowledge in the pattern base? The user combines patterns in classes $\texttt{FIBranch}_1$ and $\texttt{FIBranch}_2$ in order to identify pairs of intersecting frequent itemsets (i.e., pairs of frequent itemsets with a common ancestor in the frequent itemset lattice). To this end, the intersection join operator can be used as follows:

$$\texttt{FIBranch}_1 \bowtie^{\cap}_{\texttt{FIBranch}_1.p.s \cap \texttt{FIBranch}_2.p.s \neq \emptyset} \texttt{FIBranch}_2$$

Suppose the result is stored in a transient class $\texttt{Result}_1^{FJ}$. According to the intersection join definition, the structure of the patterns in $\texttt{Result}_1^{FJ}$ is a set containing two elements, corresponding to two intersecting frequent itemsets, one for each input class. The structure can be restructured in order to get just one set, corresponding for example to the union of the intersecting patterns, by applying the restructuring operator as follows:

$$\rho_{set\_destroy(p.s)}(\texttt{Result}_1^{FJ})$$

The result obtained by evaluating the previous expression can also be achieved by directly applying a join operation over classes $\texttt{FIBranch}_1$ and $\texttt{FIBranch}_2$, using the same join predicate and a composition function c defined as follows:

$c_{StructureSchema} = \texttt{FIBranch}_1.p.s \cup \texttt{FIBranch}_2.p.s$
$c_{Domain} = \texttt{FIBranch}_1.p.ad \cup \texttt{FIBranch}_2.p.ad$
$c_{MeasureSchema} = f(\texttt{FIBranch}_1.p.m, \texttt{FIBranch}_2.p.m)$
$c_{MappingFormula} = \texttt{FIBranch}_1.p.f \wedge \texttt{FIBranch}_2.p.f$

where f is the measure composition function also used in the intersection join (see Section 3.1.2.1).

In this case, the resulting expression would be the following:

$$\texttt{FIBranch}_1 \bowtie_{\texttt{FIBranch}_1.p.s \cap \texttt{FIBranch}_2.p.s \neq \emptyset, c} \texttt{FIBranch}_2.$$

How different are two pattern classes? The analyst is interested in determining the changes that have taken place between the patterns generated in two consecutive days. First, the user wants to find out which are the new patterns and which patterns are missing from the new pattern set. This involves a simple qualitative criterion: presence or absence of a pattern. Based on shallow equality, we take the difference between the frequent itemsets generated in the previous and the current day. If we tag the current set of frequent itemsets with *new* and the previous one with *old*, the operation for the first branch is: $\texttt{FIBranch}_1^{new} - \texttt{FIBranch}_1^{old}$. The result is stored in a transient pattern class $\texttt{Result}_2^p$.

Is the difference of two pattern classes significant? Then, the user wants some quantitative picture of the pattern set: Which patterns have significant differences in their measures as compared to their previous version? So, we need to take pairs of patterns whose similarity is above a threshold t and the difference in precision is larger than a threshold e. A simple query is formulated by the user:

$$\{(x, y) \mid x \in \texttt{FIBranch}_1^{new} \wedge y \in \texttt{FIBranch}_1^{old} \wedge \xi_e(x, y) > t$$

$$\wedge x.precision - y.precision > \epsilon\}$$

How similar are two patterns? Then the user wants to see trends: how could we predict that something was about to happen? So, he picks a new (deleted) pattern and tries to find its most similar pattern in the previous (new) pattern set, through a similarity function. Assuming the user picks pattern p from the results presented to him and the threshold is t:

$$\{\xi_e(x, p) \mid x \in \texttt{FIBranch}_1^{new} \wedge \xi_e(x, p) > t\}$$

3.3 Related work

Although significant effort has been invested in extending database models to deal with patterns, no coherent approach has been proposed and convincingly implemented for a generic model. However, since the problem of pattern management is very interesting and widespread, many researchers (both from the academic world and the industrial one) working in several different research areas are devoting ef-

forts on this. The main areas involved are standards, inductive databases, and knowledge and pattern management systems. Of course, target issues in pattern management in each area can be different. For example, standardization efforts mainly deal with pattern representation in order to achieve interoperability and pattern exchange among different systems. On the other hand, efforts in the area of knowledge management systems try to address theoretical and practical aspects concerning modeling and manipulation of patterns.

In the area of standards, there exist several standardization efforts for modeling patterns. As already pointed out, the aim of those approaches is to provide support for the interchange of data mining results and metadata between heterogeneous systems. One of the most popular effort for modeling patterns is the Predictive Model Markup Language (PMML) [PMM03], which is an XML-based modeling approach for describing data mining models (i.e., data mining results algorithms, procedures, and parameters). Its primary issue is to support the exchange of data mining models between different applications and visualization tools. The ISO SQL/MM standard [SQL01] has been proposed with the same purpose. Differently from PMML, it is based on SQL technology and provides SQL user-defined types to model patterns and several methods for their extraction and manipulation. In the programming language context, the Java Data Mining API (JDMAPI) specification [JDM03] has been proposed in order to provide data mining modeling support within the Java environment. It addresses the need for a language-based management of patterns. By adopting a JDM API a vendor may integrate its application with other existing mining applications (even if they are non proprietary) running in a Java environment. Finally, another important effort in this area is represented by the OMG Common Warehouse Model (CWM) framework [CWM01], which is a more generic modeling approach, mostly geared towards data warehousing metadata.

All these approaches do not provide a generic model capable of handling arbitrary types of patterns. Rather, only some predefined pattern types are supported. For example, all the approaches support common pattern types like association rules and clusters. Moreover, such approaches mainly address representation issues; only SQL/MM and JDM consider manipulation problems, tailored to the supported pattern types. No similarity issues are considered. Further, in some cases (for example in PMML), they do not provide support for the representation and the management of the relationship between patterns and raw data from which they have been generated.

From a more theoretical point of view, often research has not dealt with the issue of pattern management per se, but, at best, with loosely related problems. For example, the paper by Ganti et. al. [GRGL99] deals with the measurement of similarity (or deviation, in the authors' vocabulary) between decision trees, frequent

itemsets and clusters. Although this is already a powerful approach, it is not general enough for our purposes. The most relevant research effort in the literature concerning pattern management is found in the field of inductive databases, meant as databases that, in addition to data, also contain patterns [IM96, DR02]. The field of inductive databases is very active and many research efforts have been devoted to it in the CINQ project [CIN03]. The overall goal of this project is to define the basic theoretical and practical issues related to inductive query processing, which aims at extracting interesting knowledge from data. In this context, knowledge discovery is considered as an extended querying process involving both data and patterns. The CINQ project is under development and, until now, only few types of patterns have been considered (e.g. association rules, clusters, and – recently – frequent itemset [Bou04]). No arbitrary, user-defined pattern types can be represented and manipulated. However, in the future, CINQ researchers plan to consider other pattern types (e.g., graph, episode, datalog queries, etc) and provide a general inductive framework for all of them.

Our approach differs from approaches based on inductive database systems in two main aspects. First, while only association rules and string patterns are usually considered there and no attempt is made towards a general pattern model, in our approach no predefined pattern types are considered and the main focus lies in devising a general and extensible model for patterns. Second, unlike [IM96], we claim that the peculiarities of patterns in terms of structure and behavior, together with the characteristic of the expected workload on them, call for a logical separation between the database and the pattern-base in order to ensure efficient handling of both raw data and patterns through dedicated management systems.

Finally, we remark that even if some languages have been proposed for pattern generation and retrieval [MPC99, IV99, HFW+96, ESF01], they mainly deal with specific types of patterns (in general, association rules) and do not consider the more general problem of defining sufficiently expressive languages for querying heterogeneous patterns. Such rule languages are usually completely integrated in a database environment and extend the expressive power of SQL by introducing primitives for rule generation and rule querying. The result is a query language providing support for extracting patterns with a certain type. For instance, the approaches in [MPC99, IV99] deal only with association rules and clusters of association rules, whereas the approaches in [HFW+96, ESF01] deal with association rules and patterns resulting from a classification mining process.

Recently a unified framework for data mining and analysis has been proposed: the 3W Model [JLN00]. The model comprises three different worlds: the I-world, the E-world, and the D-world. The type of patterns that can be represented in such a model is the 'region', i.e., a collection of points in the space spanned by the

data. In the I-world each region is intensionally described, whereas in the E-world it is extensionally represented (i.e., for each region the members are listed). Finally, raw data from which regions are populated (by means of a mining algorithm) are stored in the D-world, in the form of relations. Such kind of patterns are very suitable to represent patterns resulting from the application of mining algorithms which split a given data set into collections of subset, i.e., classification algorithms. For example, decision trees, clusters, and frequent itemsets can be conveniently represented. Besides the model for representing patterns by using the three worlds, an algebra for manipulating patterns and data has been proposed. Such an algebra, called *dimension algebra*, provides query operators (extending the relational ones) and operations to move from a world to another. In this way, it allows one to express and manipulate the relationships that exist among a pattern, its intensional description, and the raw data set from which it has been generated.

To our knowledge, this is the first effort towards a unified mining and analysis framework taking into account formal aspects. Unfortunately, the framework is not general enough to support the representation and manipulation of arbitrary pattern types. For example, it can be shown that I-world regions slightly correspond to the approximate image we associate with arbitrary patterns whereas the E-world is a sort of intermediate mapping. However, differently from the proposed framework, no clear separation between pattern structure and images is provided, which may however simplify some types of pattern management.

3.4 Summary

In this chapter we defined a set of binary relations between patterns. We showed how these relations between patterns are reflected to relations between the sets of data that are represented by each pattern. By doing this, we have set up the basic foundation for building some a reasoning mechanism for the data. Such a mechanism, will enable the user to reason about the possible existence of data with certain properties, i.e. data that are represented by certain patterns. Moreover, we proved that the proposed set of predicates is sound and complete.

We complemented our work by presenting a series of pattern constructors and operators that utilize these predicates. The proposed operators do not constitute a fully defined query language, but demonstrate how the pattern warehouse can be used to provide in-depth information about the data collections we want to explore.

Several research issues remain open. First, the proposed query operators are a starting basis for further theoretical work. It is straightforward to prove that the operators are *consistent* with the model, in the sense that they all produce pattern classes or raw data relations as their result, even if, in case of patterns, the result

pattern type could be new with respect to the existing ones. We also conjecture (but have not proved yet) that the proposed operators are *minimal*, in the sense that no operator can be derived from the others, even if some relationships exist between them. An*axiomatization* of the proposed operators with respect to an appropriate calculus is a clear topic of future work, well beyond the scope of this thesis. Second, it is an interesting topic to incorporate the notion of type and class hierarchies in the model [RBC+03]. Third, we have intensionally avoided an extensive discussion about statistical measures: it is not a trivial task to define a generic ontology of statistical measures for any kind of patterns out of the various proposed methodologies (general probabilities, Dempster-Schafer, Bayesian Networks, etc. [ST96]). Moreover, the computation of measures for newly derived patterns (e.g., through a join operation) is also another research topic. Finally, interesting topics include the identification of appropriate structures for the linkage between pattern- and data- bases, the identification of algebraic equivalences among operators, as well as query processing and optimization techniques that take access methods and algebraic properties into consideration.

There are several publications that are related with this work; the conceptual model was first presented in [RBC+03] and it was refined and complimented by the logical model in [TVS+04b]. This Chapter as well as Chapter 2 are based on the last version of the model, which appears in [TVS+].

Chapter 4

Set values and Containment

4.1 Introduction

The model for Pattern Warehouses we presented in Chapters 2 and 3 poses a series of challenging implementation issues. One of the most intriguing research problems is the efficient navigation from the pattern to the data space and vice versa. In Section 3.2.2 we have defined operators that offer this functionality, either explicitly by using the intermediate mapping, or approximately by using the mapping formula. Both versions of the operators rely on being able to efficiently determine some simple set-containment relation: *Which patterns represent a specific record?* and *Which data are represented by a specific pattern?*. It is only natural to expect that in practice a user would like to query the *PW* for more complicated queries, involving even more complex set containment relations: *Which data that are represented either by patterns* p_1*,* p_2 *or* p_3 *are represented also by both patterns* p_4 *and* p_5*?*.

Motivated by this need of the *PW* we explored the efficient evaluation of containment queries in general. Containment queries, imply the existence of set values in our data collection. As most of modern DBMSs support the Object-Relational model [SM96], the handling of set values is becoming a natural process in a DBMS, though the usage of set valued attributes. Moreover, the efforts to integrate the handling of XML [ZND+01, KKNR04] and web documents in a DBMSs [GW00] , or to allow for IR-style queries in the database [GSBS05, FKM00], like keyword searching [HP02] impose the requirement for handling efficiently containment queries on very large collections of data.

Research in containment query evaluation in a database context has been limited [ZND+01, GW00, HM03]; the most basic results stem from the Information Retrieval (IR) and text database areas. In these areas containment is usually considered in the context of text documents in natural language. The most profound

example is the case of WWW search engines, where web documents that contain[1] the query terms provided by the user, are retrieved from a collection of millions of web pages. Research in the database field has given more focus on set-containment joins [Mam03, MGM03a, RPNK00, MGM03b, SK04, CCKN01] and similarity queries [GGK01, MCL03] and less on simple containment evaluation. Still, the efficient evaluation of containment queries in contexts like basket analysis, or bio-information processing, where the data and the queries are quite different is a challenge that database research has not addressed at its fully extent, yet. Modern object-relational DMBSs inherently support set values and queries on them, but their efficiency is lacking when dealing with the huge volumes that arise from sources like web and retail store logs or from the processing of information from bio-fields, like protein sequences and molecular structures. To this end, we propose novel indexing methods and evaluation algorithms that allow for superior performance over the current state-of-the-art for containment with evaluation.

The aim of this chapter is to give some background information and to describe clearly the research problem. We present the queries that lie in the focus of our research and the state-of-the-art evaluation techniques in detail. This chapter is organized as follows: in the rest of this section we present containment queries and describe the properties of the data. In Sections 4.2.1 and 4.2.2 we present the inverted file and the signature based indices respectively. We emphasize the former due to its importance both in applications and in research. In the next chapters, 5 and 6, we proceed by proposing two new indexing methods that provide superior performance compared with the existing approaches.

4.1.1 Data model

The focus of this work is on huge collections of low cardinality set values. Such data are typical in a variety of applications, like market-basket analysis, web statistics logs etc. Values in principle can be of any type. Throughout the thesis we assume that these are categorical values, in order to address the most general case, where we can only draw the minimum knowledge from the data space, i.e., we do not have distance metrics that allow for proximity clustering.

We focus on *huge volumes* of *low cardinality* sets, which take values from a *limited domain*. Such data are typical for transactional logs from retail stores. For example the U.S. Food Marketing Institute reports for 2005 that a typical supermarket has around $45k$ different products and reports 479 billion dollar sales for the same year with 27$ cost per transaction [Ins06]. This results in around 17.7 billion transactions in total. Moreover, it suggests a huge ratio of records per distinct item

[1]The matching criteria are more complex; still containment is a vital part of the retrieval process.

tid	product sold
1	f
1	a
2	a
2	d
2	c
3	c
3	b
3	a
4	f
4	a
4	c
5	c
5	g
6	a
6	b
6	g

tid	product sold
6	c
6	d
6	e
7	a
7	d
7	b
8	a
8	e
8	b
9	a
9	e
10	g
10	c
10	a
11	b
11	a

tid	product sold
11	e
12	b
12	d
12	c
13	c
13	f
13	a
13	d
13	b
14	b
14	d
15	e
16	b
16	f
16	a

Figure 4.1: *A collection of set values in a purely relational database*

in the database. Even assuming that the total number of different products is twice or three times that of a typical supermarket, still the ratio of the number of transactions to the number of distinct items is larger than 100.000. Given that data are collected over greater time periods than one year, and that the majority of different products remain in the market for periods larger than one year, it is easy to see how these figures become even greater. These facts assert that our assumption for large collections of set values from a limited domain are reasonable and they actually represent a very common real world case. The 27$ value of the average transaction, also suggests that in the general case the cardinality of the set values, i.e., the number of items sold in each transaction, is small. The real web logs we retrieved from the UCI repository [HB99], also adhere to the same statistical properties. For reasons of economy and clarity we introduce some notation for describing our data. Let t be a record, D be the collection of records, and I be the active domain of the database, i.e., the distinct items that appear in the D. Let also $|t|$, $|D|$ and $|I|$ be their respective cardinalities.

In order to store the transactional data from a retail store or any other source in a database, we basically have two options:

- The classical relational one, where for each item sold we insert into D a tuple $[id, oid]$, where id stands for the record identifier and oid the identifier for the product sold in this transaction. Thus, for each record t we introduce $|t|$ distinct entries in the database. An example of such relation is depicted in Figure 4.1.

tid	products sold
1	$\{f, a\}$
2	$\{a, d, c\}$
3	$\{c, b, a\}$
4	$\{f, a, c\}$
5	$\{c, g\}$
6	$\{a, b, g, c, d, e\}$
7	$\{a, d, b\}$
8	$\{a, e, b\}$

tid	products sold
9	$\{a, e\}$
10	$\{g, c, a\}$
11	$\{b, a, e\}$
12	$\{b, d, c\}$
13	$\{c, f, a, d, b\}$
14	$\{b, d\}$
15	$\{e\}$
16	$\{b, f, a\}$

Figure 4.2: *A collection of set values in an object-relational database*

- The object relational one. In this case for each record t, we introduce only one tuple $[tid, s]$, where s is a set valued attribute that contains all the identifiers for the products that were sold in this record, i.e., $s = \{oid_1, \ldots, oid_{|t|}\}$. An example of such relation is depicted in Figure 4.2.

For reasons of simplicity and readability we opt to refer to data organized according to the latter scheme; still the methods we propose in Chapters 5 and 6 work for both. The storage organization options for the data themselves are beyond the scope of this work. For storage options concerning the set valued attributes we refer the interested reader to [GMUW00]; a more detailed discussion on storage options also appears in [Ram01].

Almost all the algorithms and the index structures that exist in the literature come from the information retrieval and text databases field. The data that dominates these areas are text documents in natural language. The low cardinality set values we are addressing here, are quite different from such data. For example, techniques like stemming do not have a meaning, new vocabulary items are not introduced by new data but they are usually added separately, i.e., a new section is added to a web portal, or new products are traded by a supermarkets. With respect to containment queries the basic differences between text documents and our setting of set values are the following:

- The size of each record is significantly smaller, usually in orders of magnitude. Record entries in supermarket or in web logs, and in many other applications, contain only few different items, whereas a text document has hundreds or thousands of words.

- The number of record entries can be significantly larger. As mentioned previously the supermarket transaction in the U.S. only in 2005 amounted to 17.7 billion, which is more than the documents appearing in the whole WWW.

- There are more classes of containment queries that are meaningful in such data, hence indices should address more queries than subset.

- The vocabulary I in the context of set values is significantly limited in comparison with the vocabulary in text document collections. For example the logs of a typical U.S. supermarket would involve $45k$ distinct products whereas the vocabulary for web data can is in the order of million distinct words, as a new word appears in every 500-1000 word occurrences [ZM06].

4.1.2 Queries

We focus our research on the evaluation of simple containment queries. Simple containment queries have exact answers and they are concerned only with the existence of the query items, in the database records. Such queries have been studied in IR, in the context of the *Boolean Information Model* [WMB99, BYRN99]. Subset queries are the most important ones in information retrieval systems, and they are the basis for the ranking queries performed by www search engines. In our context we study not only subset, but also set equality and superset queries. This set of query predicates has been identified as the most important in several other studies [HM03, TBM02, MMNM03, IKO93]. Assuming a set of query items qs, which is provided by the user in each query, we define the following queries:

- *Subset queries.* In subset queries the user asks for all records t that *contain* the query set qs, i.e., $\{t \mid t \in D \wedge qs \subseteq t.s\}$.
- *Equality queries.* In equality queries the user asks for all records that *contain exactly* the query set, i.e., $\{t \mid t \in D \wedge qs \equiv t.s\}$.
- *Superset queries.* In superset queries the user asks for all records whose items *are contained* in the query set, i.e., $\{t \mid t \in D \wedge qs \supseteq t.s\}$.

For example, a subset query $Q_1 = \{t \mid t \in D \wedge \{a, f\} \subseteq t.s\}$ on relation D of Figure 4.2 would return the records with id's $\{1, 4, 13, 16\}$. An equality query $Q_2 = \{t \mid t \in D \wedge \{a, f, c\} \equiv t.s\}$ would return the record with $id = 4$. A superset query $Q_3 = \{t \mid t \in D \wedge \{a, f, c\} \supseteq t.s\}$ would return the records with id's $\{1, 4\}$.

This set of query predicates is fundamental for even more complex complex query models, like the $\mathcal{WP}$ [KST06, KKTR02], which is based on the boolean model for word patterns, and the $\mathcal{AWP}$ model [KST06, TIK05, TKD04], which extends the $\mathcal{WP}$, with *named attributes*. $\mathcal{AWPS}$ [TIK05] is an extension of $\mathcal{AWP}$, which includes a *similarity predicate*.

4.2 Indexing for Containment Queries

In research literature there are basically two main families of indexing solutions for containment queries: a)Inverted files and b) signature-based ones. In this Section

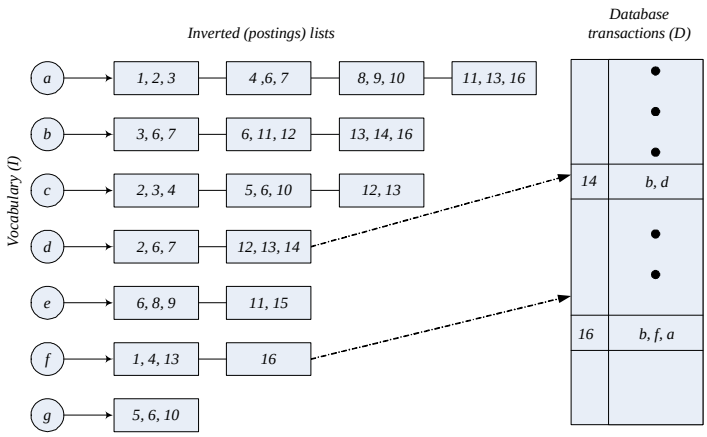


Figure 4.3: *A simple inverted file index scheme for the example of Figure 4.2.*

we review each of them and provide the basic details of their structure.

4.2.1 Inverted files

Inverted files [Knu73, BYRN99, WMB99, ZMSD92] are the indexing solution employed by all WWW search engines and the most efficient index structure for text query evaluation in research literature [ZM06]. Inverted files comprise a collection of lists, one per item of the database, recording the identifiers of the records containing that item.

As we can see in Figure 4.3, an inverted file index has two major components: (a) the *vocabulary* and (b) the *inverted lists*. The vocabulary is a list of all the distinct items appearing in the database, i.e., it is the same with the database vocabulary I of Section 4.1.1. Each list node has a label indicating the item it represents and a pointer to the head of the inverted list. The vocabulary is usually organized in a B-tree or a trie, if range queries are of interest, or in a hash table, if they are not. The inverted list contains information about all the records in which the item appears. For each appearance of an item in some record, there is a distinct posting in the respective list. Each posting contains at least an identifier for the record and in the case of text documents the multiplicity of the item's appearances in the document, the offset of each appearance, etc. In our case, this information comprises the record *id* alongside with its length. The length of the record is required in order to efficiently execute equality and superset queries. Inverted lists for document collections are usually sorted according to the document *id* in order to provide an efficient intersection implementation, by merge-join. Still, there are other techniques, where the postings are sorted according to their in-document frequency [WL93, Bro95] or their impact [AdKM01, AM02]. These alternative organizations of the inverted lists can offer superior performance in certain kinds of queries. This is achieved by pro-

Item	Signature
a	0000 0001
b	0000 0010
c	0000 0100
d	0000 1000
e	0001 0000
f	0010 0000
g	0100 0000

Table 4.1: *Signatures of items*

viding approximate answers, using only the first part of each inverted list. Since we target boolean queries with exact answers, we assume that the lists in the inverted files are sorted according to the record *id*.

Keeping this kind of information in each posting, may result in huge lists; the *id* and the length of a transaction is inserted as many times as the number of items it contains. This means that, theoretically, the size of the inverted file could be similar to the size of the database or even larger. In Section 4.6 we examine techniques to compress the list size in order to improve evaluation time.

4.2.2 Signature files

Signature files [Fal92, FC84, Fal85, FC87] are the basic alternative method to the inverted file and traditional RDBMS methods for containment queries. The main idea behind signatures is to hash each item of I to a fixed size word and then to *superimpose* the codes of all the items of a record, to create the records' *signature*. Superimposing relies usually to the usage of some some binary operation (AND, OR, XOR etc) between two signatures. For example in Table 4.2.2, we depict the record level signatures for D; to construct them we used OR as the superimposing function and the signatures of Table 4.2.2 for the items of I.

The basic elements of the evaluation procedure are common for all the signature methods; the signature of qs must be produced, and the *candidate* results must be retrieved from the signature file. As in all signature based methods to provide the final results, all candidate records must be retrieved and tested. The testing is necessary because of the possible *false positives*. In order to reduce the size of the signatures, not every combination of items and set-values has a distinct signature. Thus, it is possible that the signatures of some records might match the signature of qs, but the records might not. These records are generally referred to as false positives.

A straightforward index structure that takes advantage of the signature methods is the *sequential signature file* [IKO93]. The idea is very simple; the record signa-

tid	products sold	signature
1	$\{f, a\}$	0010 0001
2	$\{a, d, c\}$	0000 1101
3	$\{c, b, a\}$	0000 0111
4	$\{f, a, c\}$	0010 0101
5	$\{c, g\}$	0100 0100
6	$\{a, b, g, c, d, e\}$	0101 1111
7	$\{a, d, b\}$	0000 1011
8	$\{a, e, b\}$	0001 0011

tid	products sold	signature
9	$\{a, e\}$	0001 0001
10	$\{g, c, a\}$	0100 0101
11	$\{b, a, e\}$	0001 0011
12	$\{b, d, c\}$	0000 1110
13	$\{c, f, a, d, b\}$	0010 1111
14	$\{b, d\}$	0000 1100
15	$\{e\}$	0001 0000
16	$\{b, f, a\}$	0010 0011

Table 4.2: *Signatures for set-values of Figure 4.2*

tures are stored sequentially in a separate file from the original data; when a query is issued, the signature of qs is generated and the signature file is sequentially scanned and compared against the qs. The records whose signatures match[2], are retrieved and tested to produce the final results. An obvious weakness of this indexing scheme is that the whole index has to be scanned when a query is issued. To tackle this problem *Bit-Sliced Signature Files* were proposed [IKO93], which store the signatures in columns, i.e. the store the first bit of all the signatures of D together, then the second and so on. In this scheme it is no longer required to scan the whole index. Scanning only the first bits of each signature might be enough to locate a small group of candidates. Other methods that allow access to only a part of the signature file, through different clustering strategies [Hel97, SDKR$^+$95].

One step further than grouping is the idea of signature trees [Dep86, AWY99, NTCM02, TBM02, MCL03, Che05]. Signatures trees organize the records' signatures following the same lines as R-trees [Gut84, SRF87, BKSS90, The03] and B-trees [Bay71, BM72]. Each node has a signature, which is the result of superimposing the signatures of its immediate descendants. The idea is compelling; given the signature of qs, the evaluation algorithm can find the node(s) that match it at each tree level and subsequently examine it subtree, until the matching candidate record is found. Moreover, with the suitable splitting and merging algorithms, the tree can be balanced, thus its height can remain small even for large collections of records. In [NM02] the authors propose two additional enhancements for signature trees: a) they propose an alternative splitting algorithm, which provides a more efficient grouping for signatures into tree nodes and b) a paging scheme which the contents of several logical tree nodes in one physical page. This paging scheme allows the existence of relatively small logical nodes, which offer a superior pruning power, as a result of limiting false positives. The results of these two enhancements is the superior behavior of the resulting signature tree over the classical signature tree

[2]The meaning of match depends on the query predicate (subset,superset etc) and of course on the superimposing function.

for similarity queries. An interesting approach that outperforms the signature tree for subset and superset queries in proposed in [TBM02], where the authors propose combining the signature tree with parametric weighted filters. By coupling the PWF with the signature tree splitting techniques they create three new PWF variations, with performance gains up to 85%.

Still, signature based methods in a series of comparisons [CBM+94, ZMR98, HM03] have been shown to be inferior to the inverted files, except maybe for set-equality [HM03] and similarity queries [MCL03]. In [CBM+94] a comparison in the context of a parallel system between signatures files and inverted files, favors the latter; in [ZMR98] the authors show both analytically and experimentally that inverted files significantly outperform signature files for boolean queries over text. These results are verified experimentally in a context similar to ours in [HM03]. In [HM03] Helmer and Moerkotte evaluate, using the queries we presented in 4.1.2, the inverted file and three signature based methods the sequential signature file, the signature tree and the extendible signature hashing. The basic problem with signature based methods is the number of false positives. In any setting, the cost of retrieving a false positive from the disk is one of the most expensive operations during query evaluation [ZMR98]. But avoiding having false positives is a very hard task, since one has to give too many bits to the different items of I. Moreover, since we do not query the database just for set quality (which could be solved easily with a hash) but also with subset and superset predicates, we would like have $2^{|I|}$ different signatures for any possible combination of the items of I. Even if we have a limit on the size of the record size $|t|_{max}$, the possible set values are $\sum_{n=1}^{n=|t|_{max}} \binom{n}{n}$, which is still huge. A smart choice of the hash function for the items of I and of superimposing function, can limit the effect of the huge domain, but still the problem exists. Superimposing makes it even worse, since at higher tree levels (near the root), where there are numerous items under each node, the node signatures tend to become similar. This can be leveraged by grouping together similar signatures [Dep86], but still the experimental results have shown, that even for low cardinality sets [HM03], inverted files are superior.

An in-depth study of how containment queries are evaluated by inverted files and RDMBSs in [ZND+01] demonstrates that the RDBMS usually performs worse, often by orders of magnitude. RDBMSs are able to outperform the inverted file only when the selectivity of the query predicate is very small.

Since inverted files appear as the state-of-the-art index for containment queries we dedicate the rest of this chapter in detailing several aspects of their behavior and we explain how we adjust them in our context.

4.3 Query Evaluation with Inverted Files

The evaluation of the subset, equality and superset queries against inverted files is rather straightforward; in all cases we need to scan the inverted lists of the items that appear in qs once, in order to retrieve the correct answer. In the case of subset and equality queries we have to intersect the inverted lists and in the case of the latter we use the length as an additional filter. To evaluate superset queries we take the sorted union of the lists and use the length again as a filter.

The algorithms we present here differ slightly from the algorithms used for evaluating the same queries in IR. The basic difference is that we process the inverted lists simultaneously, by doing k-way merges and k-way merge joins. In IR the algorithms usually perform a sequence of 2-way merge joins between the current candidate solutions and the inverted list of one of the remaining items of the query set qs [MZ96, AM98]. The k-way merge join is theoretically more efficient since it might stop scanning the inverted list sooner, since it stops when it reaches the end of any of the involved lists. Still, in the context of IR this is not an important factor for two reasons: a) the most popular IR queries, the ranking queries are evaluated like disjunctive queries, i.e. a document cannot be easily discarded and usually the whole lists of the qs items have to be scanned and b)inverted lists in a text document context are usually smaller than the ones we expect in a transactional log context, since the most popular words in text documents are "stop" words, which are not considered in queries. This way performing sequential 2-way merge joins by retrieving the whole lists into main memory does not introduce a significant overhead compared to performing a k-way merge join. Moreover, using list of main memory candidates is imperative for the evaluation of ranking queries that need to update the score of each candidate during the evaluation. Performing sequential 2-way merge joins, and keeping the results as *candidates* in main memory can provide benefit only if we have queries with big qs and random access in the inverted lists. Having a big qs makes probable that during query evaluation the number of candidates will decrease significantly, thus being able to randomly access the larger lists to examine whether they contain a candidate or not might be compelling.

Keeping the candidates in main memory provides a straightforward algorithm for evaluating boolean queries; all the lists of the qs items are retrieved sequentially, and candidates are discarded if they fail to meet the specific matching criterion, e.g. for subset queries they are discarded if they do not appear in one list. Still, in a context were $|I| \ll |D|$ the keeping of candidates into main memory is impractical, since the inverted lists can be very large. This is the main reason why the query evaluation algorithms we present here, do not keep candidate solutions in main memory but instead they need only a vector of qs elements. This vectors holds a posting and a

Subset Query Evaluation

```
// qs = {o_1, ..., o_n}, o_1 < ··· < o_n.
1.  for each i in the inverted list of o_n, o_n.list {
2.    if |qs| < i.length continue
3.    for (j = n − 1; j ≥ 1; j − −) {
4.      found=false;
5.      while (c < i) {
6.        get next item c from o_j.list;
7.        if (c == i) then found=true }
8.      if (not found) break }
9.    if (found) output i; }
```

Figure 4.4: *The subset query evaluation algorithm for the inverted file*

pointer for each inverted list of the qs items, to facilitate the k-way merge join. In the following we present in more detail how each algorithm works.

Subset Queries. In the case of subset queries we merge join the inverted lists of the items that appear in qs. Assume the database depicted in Figure 4.2 indexed by the inverted file presented in Figure 4.3 and a subset query with $qs = \{a, f\}$. The evaluation algorithm retrieves the inverted lists of the items a and f, and since they are sorted, it merge-joins them, to trace id's which have postings in both lists. i.e., $1, 4, 13, 16$. The algorithm is depicted in pseudo-code in Figure 4.4. The algorithm starts scanning the lists from the smallest to the largest, in order to increase the chances of quickly eliminating as many candidate id's as possible. For example, if the user asked for the records that contained b, f, g, starting from the inverted list of g would not require scanning more that the inverted list of g and the first disk page of f's inverted list. On the other hand, if we started from the inverted list of b, which is significantly larger than the list of g, we would have to scan the three disk pages of the inverted list of b, the first disk page of the inverted list of f and the inverted list of g.

Since the records in our context are relatively small, and queries retrieve all exact matches, we can use the length of the record as a heuristic to speed up processing; we can discard records that have length smaller than $|qs|$ right away.

Equality Queries. The evaluation procedure for equality queries is almost identical to the evaluation of subset queries. The basic difference is that the use of the length of each transaction as a filter is imperative as a filtering step. Moreover, the filtering condition is stricter; only transactions whose length is equal to $|qs|$ are eligible for further consideration.

Superset queries. Superset queries are not common in information retrieval, and to the best of our knowledge there is no algorithm for evaluating them by using inverted files described in detail. In [TPVT] we have used an algorithm that

Superset Query Evaluation

// $qs = \{o_1, \ldots, o_n\}$, $o_1 < \cdots < o_n$.

```
1.  for each i in the inverted list of o_n, o_n.list {
2.    if |qs|¡i.length continue
3.    for (j = n − 1;j ≥ 1;j − −) {
4.      counter=1;
5.      while (c < i) {
6.        get next item c from o_j.list;
7.        if (c == i) then counter = counter + 1 }
8.    if (i.length == counter) output i; }
```

Figure 4.5: *The superset query evaluation algorithm for the inverted file*

progressively traces all the combinations that contain each item of qs. Such an algorithm is also implied in [HP94], but no details are given. Whereas this algorithm required scanning more that once the inverted lists of the items of qs[3], the algorithm we present here requires a single scan of each list. The basic idea is that if we have the union (assuming bag semantics) of the inverted lists of qs we can easily determine which records satisfy the superset predicate by using the length of each transaction; all those, whose length is equal to the multiplicity of its postings in the union of the inverted lists, satisfy the query. It is easy to see that if the length of a record is greater than the number of times it appears in the inverted lists of qs, then more items than those of qs must be contained in this record, thus it cannot satisfy the superset predicate. Moreover, by creating the union sorted, we never need to "remember" anything more that last $|qs| + 1$ postings to determine if a record satisfies the query predicate. The algorithm in pseudo-code appear in Figure 4.5

The previous algorithms rely on retrieving and merging the inverted lists of the items that appear in qs. As a result their performance deteriorates significantly when these lists grow very long. Unfortunately in a setting like the one we described in Section 4.1.1, where the size of the database is very big and the number of distinct items is small, we cannot avoid the phenomenon of queries that have to handle very long lists. Moreover, even if the number of items is not so large, and the average list has a moderate size, problems arise in practice, since the distribution of the items in most real world activities is skewed. This means that the most frequent items can have very long lists, even if the average list is moderate. In addition, database experience indicates that more frequent items are the ones that appear more frequently in users queries.

Using an index to provide random access to the inverted lists does not provide

[3]Actually it scanned once the longest inverted list, twice the second and so on.

significant benefits. It does not have any impact on superset and its effect on equality and subset queries is limited, except if the selectivity of the query is very small. When performing a merge join between two or more sorted lists, we slide positional indices in the lists from their beginning towards their end, checking if a value occurs in all lists. Using an additional index structure on the lists to check if a value exists in one of them, would only be beneficial if we could guess successfully the place where the value appears in the list. If the value we are searching for, is very close to the current offset in the list, it is better to just fetch the next few disk pages, than to use the index to randomly access the list. Only if we knew that we have to fetch many pages before getting to the place where the value should appear, it would be beneficial to randomly access the list. Still, it is not easy to make such predictions accurately. For this reason, to the best of our knowledge there are no research works that use indices on the inverted lists of an inverted file index in order to speed up the evaluation of boolean queries. The only exception we are aware of is [MRYGM01], where an index is built as a result of the breaking up of each inverted list to fixed size buckets, in its BerkleyDB implementation. But since using the index is not in the focus of the paper, it does not give any results comparing evaluation processing with and without the index, so no conclusions can be drawn.

The problem of large inverted lists has been mainly addressed in research by using *compression* techniques; a brief survey of the state-of-the art is presented in Section 4.6. Moreover, the possibly large size of the inverted lists has several implications in the storage allocation and maintenance strategies. We examine these problems in Sections 4.4 and 4.5.

4.4 Storage Allocation Strategies

The most effective physical storage allocation scheme for inverted lists is to store them contiguously on the disk [ZM06, TGMS94, LZW06].

Alternatives in research literature and in practice are motivated by the difficulty to support contiguous storage in the case of updates. Moreover, another issue when dealing with storing inverted lists of different sizes is the efficient space utilization. To deal with these two issues several different storage allocation strategies have been proposed. Usually, a basic point in these strategies is to treat long and short inverted lists differently. Having lists that vary substantially in size, as a result of skewed item distributions in the database, which is a common real world case [ZM06, BCC94, TGMS94].

To deal with updates efficiently, alternative approaches either overallocate space for future updates or split the long lists of the inverted file in smaller chunks of disk pages, stored contiguously in the disk. In [BCC94] fixed size objects, whose size

ranges from 16 to 8192 by powers of 2, are allocated for long lists. When the list fills the 16 bytes, then a chuck of 32 bytes is allocated and the initial contents are copied in the latter chunk. When the maximum size is reached, instead of copying the contents to a new large chunk, a linked list of the maximum size chunks is started. The authors report a deterioration in query performance, with respect to a contiguously stored inverted files, from 6% to 29%, but at the same time they provide a superior update time. An analytic approach in [LMZ05] shows that the cost on query performance might be larger in the general case. In [TGMS94] long lists are kept initially contiguously in the disk and updates are stored in different buckets. The authors report that this allocation scheme reduces the performance in half. Lester et al. [LZW06] report observing such behavior when the inverted lists are split in only two pieces. In [LMZ05], the authors present a technique of keeping large lists partitioned in linked chunks whose sizes follow a geometric sequence. By using periodic mergings the authors are able to keep a controlled number of partitions for each lists, thus avoiding significant fragmentation. The authors report a query response time greater by 20% from the respective evaluation time against the contiguous list scheme when using 2 partitions. In [MRYGM01], the authors explore the embedded implementation of an inverted file in a DBMS, the BerkeleyDB. This case is of special interest in our setting since the storage and indexing of the data in a DBMS would enable the easy handling of the set-valued data together with other semantically linked information. Melnik et al. explore three different organization schemes for the inverted lists in the DBMS: (a) a scheme where each list comprises a single entry in a table, with the respective item as a key, (b) a scheme where each posting in a list comprises a single entry in a table, with the item and the record *id* forming the key, and (c) a mixed approach where lists are split (or merged) to fixed sized buckets; each such bucket is a distinct entry in a table with the item and the the last record *id* of the bucket forming the key. Experimental results demonstrate that, if rightly tuned, the mixed approach can have a query evaluation performance very close the first approach and a significantly better update time.

The problem of space utilization arises when there are numerous short lists that must be stored efficiently. A basic technique to address this problem is to treat long lists and short lists differently. In [TGMS94] and [MRYGM01] short lists are merged in buckets of fixed size in order to increase the utilization of reserved space in disk. In [CP90] short lists are kept together with the vocabulary in a B-tree, whereas long lists are kept separately. This approach offers increased performance when dealing with short lists.

Finally techniques for creating a distributed inverted file index appear in [MRYGM01] and [JO95].

4.5 Creation and Maintenance

The creation and maintenance of the inverted file pose a series of interesting problems, mainly due to the large size of the datasets that must be indexed. In the following we describe effective algorithms for creating large inverted files under limited memory and examine maintenance algorithms both for contiguous and non-contiguous inverted lists.

4.5.1 Construction

The construction of an inverted file for a small dataset, where everything can be done in main memory does not pose any complicated problems and it is rarely a practical problem. There are three main strategies for creating an inverted file [ZM06, SH03]:

- *Two-pass inversion* [FL91]. Following this strategy, the database is scanned twice; in the first pass the algorithm collects statistics, like the frequency of item appearances, the total number of records etc, in order to allocate the right amount of space in memory for the building of the index. In the second pass the actual postings are created and inserted in each inverted list. In [MB95, WMB99] this method is adjusted for working under limited memory by allocating the right space in the disk after the initial pass and flushing the main memory part of the index to the disk, when the memory is full. The basic weaknesses of this method are that it requires scanning and parsing the data twice and that it has to sustain an array equal to the vocabulary size, in main memory.

- *Sort-based inversion.* Sort-based inversion has been used in several research works [CCH92, BCC94, HFBYL92] and its basic idea is that from a single scanning of the data, files with postings are created in the disk, which are subsequently sorted and used for creating the inverted file. A detailed description of such an inverted file creation scheme is given in [MB95]. In this creation method, as data are scanned and parsed the respective postings are created, compressed and stored into main memory. When there is no more available memory the postings are sorted, first by item and then by document *id*, and they are flushed to the disk. Each such sorted chunk of data is referred to as a "run". When all data have been processed the inverted file is created by the scanning the runs once. A weakness of this method is that it might require temporal disk page equal or greater than the size of the final inverted file. Still, the authors of [MB95] present a method by which the final inverted file can be created on the space that is freed as each run is processed. Advantages of this method are the fact that it needs only one pass on the data (but it needs

to write and read all the runs) and that it uses compression to make the usage of the available memory and disk space more efficient.

- *Single-pass inversion* [SH03]. The single-pass inversion scheme combines elements from both the previous methods. It scans the data only once like the sort-based inversion and creates parts of the inverted file as the two-pass inversion in main memory. The parts are flushed to the disk and then merged to create the final inverted index. An advantage of this method compared to the other two is that it does not need to keep all the known vocabulary in main memory, but only the part of the vocabulary that came up in the current run. This way, more memory is available for the creation and less runs needed.

Note that all the previous strategies aim at creating the optimal organization for the inverted index, i.e., contiguous inverted lists. If it is acceptable that the lists can be broken in pieces as in [MRYGM01], the handling of the runs might be more straightforward, as distinct chunks can be created in a single run, without the need of processing them again. Another point with respect to the context of this work, is that these comparisons come from experiments on text (web mostly) data, in [SH03] and other works in IR. As a result there are several differences with the kind of data we are addressing in this work. In the aforementioned methods the handling of the vocabulary (it must be all in main memory or not) was an issue, since the vocabulary grows linearly with the size of the data in text documents. On the other hand, the small (wrt to the database size) vocabulary we assume here, does not play a significant role in the creation procedure. Another important point is the cost of scanning the data vs. the cost of writing and reading runs from the disk. We expect that reading and parsing small transactions with no item repetitions and without keeping complicated statistics will be significantly easier than reading and parsing text documents, where the vocabulary is not known a priori and the size of each document is many times the size of each transaction. These factors favor the two-pass inversion scheme, thus the actual difference between the other two methods can be insubstantial.

4.5.2 Maintenance

At a first glance updating inverted lists seems an easy task; when new documents/records arrive they are assigned *id*'s greater that the ones assigned to already indexed records, and the respective postings are added at the end of the inverted lists. In practice there are several problems, stemming from the storage allocation schemes that are used. The most efficient organization for the inverted lists is to store them contiguously on disk. Such a scheme is not trivial to maintain in the presence of continuous updates; overallocation strategies, even when statistics are

used [SC05] have their limits. In the following, we present the most popular methods for maintaining the inverted files in the presence of updates.

The need to update an inverted file is usually the result of new document additions in the database. Still, deletions [CCB94, CCB94] and internal changes in the documents [LWP+03] have been considered. All strategies that require that the inverted lists are stored contiguously, store the updates initially in a *main memory inverted file*. In modern hardware it is easy to hold information for numerous documents in a main memory inverted file, especially if an efficient compression method is used. The benefit of this approach is that new records are immediately available for querying, whereas the update cost for the main memory inverted file is very small. When the available memory gets full, it is necessary to move all the postings from the main memory to disk. This way we only do batch updates in a frequency depending on the rate of updates and the available memory buffer. Caching strategies for reducing update costs are considered in [BMV96]. There are three basic approaches for updating the disk-based inverted file, when the memory buffer is full [ZM06, LZW04]:

- *Re-build.* The simplest method for updating the index and keeping the inverted lists stored contiguously is to re-build it from scratch. This is also motivated by the fact that many IR systems can afford to be periodically off-line for a short time. In [MRYGM01] it is stated that most commercial web search systems employ this method for efficiency and simplicity.

- *Re-merge.* In the re-merge update scheme, the main-memory and the disk based inverted file are merged and a new inverted file is written on disk. This requires a single scan on the index. The construction time is reported [LZW04, LZW06] to be significantly less than in the other two methods. During the re-merge the old index may still be available for querying, but no new documents can be indexed at the same time.

- *In-place.* The basic difference between in-place and re-merge approaches, is that the in-place does not merge the whole main-memory and disk based inverted files, but rather specific lists, e.g. the longest ones. This strategy can be combined with other actions of the system, for example the update might take place when a list is retrieved for querying purposes; since the list is already retrieved it can be merged with the respective list of the main memory index and then written again on disk. [LZW06] reports that this approach may be attractive for very large systems, but in the general case the re-merge strategy remains more effective. In a context like ours, where the vocabulary is small, we do not expect that this update strategy will be fruitful, since even small batches of new records affect a large part of the vocabulary.

The pervious update techniques refer to a storage scheme where the inverted lists are stored contiguously in the disk. In addition, there are several proposals where the lists are stored in several partitions in the disk. In such cases the merging stage is a lot easier. If the number of partitions for the inverted lists is not controlled or limited [TGMS94, MRYGM01, BCC94, STGM94] then it is possible that no merging at all is needed; the new postings are stored in a new partition on the disk, and a link is added at the end of the previous partition to the new one. Even if the number of partitions is limited [LMZ05], we only need to merge a part of the disk-based inverted file with the main-memory one. The trade-off in each case is that query performance deteriorates as the inverted lists get fragmented.

A technique of periodic merging is introduced in the INQUERY system [CCB94, CC95], where a part of the main memory inverted file is periodically merged with a part of the disk based inverted file. This way the index rotates in the available disk, with the updated part written in the available free space, and the obsolete part of the index becoming available as free space.

Finally in [LWP+03] the case of changes in already indexed documents is considered. The proposed technique addresses *in-document* changes and focuses on preserving the information on word positions in documents. To avoid unnecessary updates to items that are already indexed, the authors propose tracing the offset of an item in a document, relatively to a *landmark*. This way, changes that occur in the document before the landmark, would not affect the offset information about the word.

4.6 Compression Techniques

In Section 4.3, we saw that the evaluation of queries relies on merging the inverted lists of the qs items, and as a result it is highly dependent on the size of the lists. The basic technique for leveraging this problem is to *compress* the inverted lists [SWYZ02]. This way, space requirements and disk page accesses during query evaluation are reduced by paying a small CPU overhead for the coding and decoding of the compressed data. The basic idea behind compression is to take advantage of the fact that queries access the inverted lists sequentially and represent the record id by using d-gaps, i.e. their difference from the previous id. For example if the inverted list of item a is the following:

$$a : 1,\ 2,\ 3,\ 4,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 13,\ 16$$

by using the d-gaps it would be represented as:

$$a : 1,\ 1,\ 1,\ 1,\ 2,\ 1,\ 1,\ 1,\ 1,\ 1,\ 2,\ 3$$

Now, the inverted list of a is a sequence of smal numbers that are frequently repeated. Actually, the longer the inverted list is, the higher the density of the record *id*'s will be, thus the d-gaps will be smaller. If we use some fixed size representation for numbers, for example 4 bytes, the storage requirements for both lists will be the same. The idea of compression lies in using some *variable bit representation, which is efficient for small numbers*. Even something as simple as unary or binary codes would significantly reduce the space needed to store the inverted list of a, when represented by d-gaps. Popular bitwise compressions schemes are Elias gamma and delta codes and Golomb-Rice code [Gol66], with the latter being the most effective compression scheme used [SWYZ02, MZ96, AM98, WMB99]. The effectiveness of the Golomb-Rice encoding is that it can be parameterized so that the most frequent d-gap values is compressed most effectively. The idea behind the Golomb-Rice encoding is to divide the number by the chosen *divisor* (this is the parameter of the algorithm) and then code the quotient in unary and the remainder in binary.

Compression can reduce the size of the index by a factor of 3 to 6 using the previous techniques. At the same time it introduces a CPU overhead for the coding and decoding the inverted lists. To leverage this problem there have been some works on more lightweight codes, which perform the encoding at the byte level [WZ99, SWYZ02, CM05]. The idea is simple; values smaller than 128 are packed in one byte with a "0" leading bit. If the value is larger, then it is divided with 128 and the remainder is coded in one byte, with leading bit "1", and the quotient is recursively encoded in the same way. Related studies have shown that bytewise encoding introduces a small size overhead but it is considerably cheaper computationally, making it an overall attractive choice.

A consequence of compressing the inverted lists is that every list must always be read sequentially in order to be able to uncompress it. This is because every *id* can only be computed as the sum of all previous d-gaps. As we have already mentioned in Section 4.3, when the qs is large there are cases where the candidates that appear during query evaluation are significantly limited. Therefore, uncompressing a whole inverted list when we need only to check for the existence of few *id*'s seems a waste. In [MZ96] and [AM98] the authors propose using synchronization points that enable skipping the decoding of several parts of the inverted list. In [MZ96], the authors propose to introduce a *synchronization* point at every c number of entries in the inverted list. A synchronization point comprises the *id* of the last record that appears in the list (not only a d-gap from the previous) and a pointer to the next synchronization point. This way, the evaluation algorithm can jump from one

synchronization point to the other, skipping the decompression of the intermediate values if it can infer that no candidate can be among them. The gain in *CPU cost* can be reduced for boolean queries of 5-10 items by 80%, for a 20% overhead in the list size. A similar method is proposed in [AM98], where the domain of the record *id* is divided in equal size partitions and the record *id*'s are represented as *fixed size* *d*-gaps relatively to the start of each partition. Moreover, in the begging of each inverted list is marked how many of the records of this list that fall in each partition. Using this information and the fact that *d*-gaps are encoded in fixed size binary code, one can trace the offset of the record *id*'s that fall in each partition, in each list.

Chapter 5

The Hybrid Trie-Inverted (HTI) File Index

5.1 Introduction

Inverted files are the state of the art index for containment queries, as we showed in Chapter 4, but still their performance suffers when the inverted lists become very long. This is due to their internal structure: inverted files contain a header list with all the items of the vocabulary; for each item, an inverted list with pointers to the records that contain it is maintained. Thus, if some items appear in many set values as a result of a huge database with items from a limited vocabulary or/and with skewed distribution, their inverted lists become very long. The evaluation algorithms for containment queries that were presented in Section 4.1.2 usually scan the entire inverted lists of the qs items, thus the existence of long inverted lists has a deteriorating impact of the query evaluation. Unfortunately, this is a common case in the real world. In data from supermarkets, an average item participates in hundreds of thousands of transactions in a single year. The most frequent items probably participate in millions of transactions. The same picture appears and in the real web log data we came across in the UCI KDD archive [HB99]. These datasets are logs that trace the behavior of users in large web portals, which is a common source of data that are analyzed by using containment queries (e.g., *"Which users downloaded only drivers and patches from our website and did not visit any other page?"*). In these datasets, a few popular web locations from the portal, appear in numerous user sessions, and the total number of records is large even for very small time periods of tracking user behavior. As expected, the inverted file does not behave well in such data, and new methods for answering containment queries are required.

To efficiently address this type of queries, we propose a novel indexing scheme,

the Hybrid Trie-Inverted file (HTI) index. The basic idea behind the HTI-index is to break up the larger inverted lists, to smaller ones, in a way that facilitates efficient query evaluation. Since, queries require knowledge about the combinations of different items in database records, we follow the same line when breaking up the lists; each new list is constituted by the *id*'s of records that contain a specific combination of items. From a structural point of view, the HTI-index superimposes a trie structure, the *access tree*, over an inverted file index. The access tree offers pointers to intermediate points in the inverted lists of the most frequent items. Practically, it breaks up the long inverted lists, thus leveraging the performance of inverted files. The evaluation of queries is performed partly in the access tree, for the most frequent items, and partly in the inverted file for the rest of the items. This setting greatly reduces processing time and, at the same time, the memory requirements remain low since the information for the vast majority of the data is kept in the inverted file. The average impact on query answering efficiency is further amplified if, as it is commonly assumed, database items are queried according to their frequency of appearance. In short, our contribution comprises the following:

1. We propose a novel indexing scheme, the HTI index that combines a trie with an inverted file, for large collections of low cardinality sets. The main idea is that the trie is placed in main memory, indexing the $top - k$ most frequent items of the data set, whereas the inverted file is placed in secondary storage, associating each item with all the records that contain it. The index is particularly fit for data from a limited domain or skewed data, which is a very common real world case.

2. We present efficient evaluation algorithms for set containment queries that utilize the proposed index. For all types of queries, we evaluate the part of the query that involves frequent items by exploiting the main memory part of HTI, and complement the answer by testing the infrequent items through the inverted file.

3. We demonstrate the superiority of our proposal over the state of the art access methods, by extensive experiments. We evaluate the HTI index on real and synthetic data. We assess the number of disk page accesses performed by the HTI index as a function of domain of items, database size and size of the query set. In all occasions, HTI significantly outperforms a competitor inverted file, and scales gracefully, especially in the cases of large database and query sizes (as opposed to the inverted file that fails to scale similarly). In the case of the real datasets, which involve 320k and 1M records, the HTI index performs an order of magnitude less disk page accesses with a memory overhead of less than 0.5MB. Our experiments with synthetic data show that

even for large domains, keeping a low threshold for the top-k items held in the trie is sufficient for achieving high performance with minimum memory expenses.

To the best of our knowledge, this is the first time that a trie is used as an index for evaluating containment queries. Although our study is specifically focused on set-valued database attributes, we anticipate that its applicability is wider and reaches other cases, like text indexing, too.

5.2 The Hybrid Trie-Inverted file index

5.2.1 Index structure

Tries and inverted files have been extensively used in the context of text databases, but the former have not been employed for indexing set valued attributes in object-relational databases. In this section, we introduce the HTI index that combines a main memory trie with an inverted file residing in secondary storage. First, we explain how a trie can be used to to keep track of different combinations of items.Then, we show how the trie and inverted files are combined in the HTI index. Finally, we also discuss issues concerning the creation and maintenance of the HTI.

5.2.1.1 Grouping by item combinations

Set values by definition do not provide any ordering for their internal items, thus we can order and store them in the most convenient way. This is very helpful if we want to keep information about combinations of items, since we can decide on a common order for storing each item and each combination of items. For example, the set-values $\{g, a, c\}$ can be always stored as an alphabetical sequence $\{a, c, g\}$. This way we avoid any ambiguity that might arise from set representation and moreover we can provide more efficient storage for common combinations of items. By representing each combination of items as a sequence we can use a *trie* to store efficiently each different sequence.

Tries are multiway tree structures for storing string keys which enable retrieval in time proportional to the string length [BYRN99]. Unlike inverted files, tries are letter oriented and each string corresponds to a path in the tree. Consequently, common prefixes in strings correspond to common prefix paths in the tree. Leaf nodes include either the documents themselves, or links to the documents that contain the string that corresponds to the path. Since strings are words of some language, the maximum number of children for a node is limited by the number of letters of the alphabet of the documents' language. The way tries are created allows

for prefix (or suffix, if strings are inverted before being mapped to paths) search, i.e., they provide a kind of range search, based on the first letters of the string.

A significant difference between set values and text documents is that, unlike words (which are composed of letters), the items of a set-values are not further decomposable to smaller units. Even if the items are alphanumeric values themselves, this is simply a coding scheme of the database, that eventually has no relationship to the user queries. Therefore, it is meaningless to exploit the alphanumeric value of the items for indexing purposes, but rather we need to use the set of all items I as the vocabulary of the index. As a result, each node might have $|I|$ intermediate descendants. This makes the potential size of the trie very large and thus reduces the gains in terms of space, achieved from common prefixes. Still, choosing the *frequency of appearance* as the basic ranking function for each item of I, we can considerably reduce the space requirements of a trie. Following such an ordering for I, we expect to find, in the first places of each sequence, the items that appear in the database more frequently, thus there is increased probability that several sequences will share the same prefix. Since common prefixes are stored only once in a trie, this will result to small tries for storing the combinations at hand.

To construct a trie for set values, we follow the approach of Han et al. in [HPYM00, HPYM04]. First, each record is transformed from an unordered set to an ordered *sequence* based on the item frequency. An item x precedes another item y in an ordered record if x is more frequent than y in the whole database D. The ordered record is subsequently mapped to a path starting from the trie tree root. If some nodes already exist, due to a common prefix with a previously inserted record, we only add the new nodes.

An index based on a complete trie tree for the database of Figure 4.2 is depicted in Figure 5.1. The record with $id = 4$ and set value $s = \{f, a, c\}$ is ordered according to the frequency of its items in the database. Since a occurs 12 times, c occurs 8 times and f occurs 4 times, the record's set is transformed to a sequence $s = \{a, c, f\}$ that subsequently contributes the path $a \rightarrow c \rightarrow f$ in the trie. Using this structure for indexing is straightforward; each node provides a link to a list that contains the id's of the records whose sequences have as prefix the path from the root to the current node. This means, that all the records that are identified in the list contain all the items that appear in the path from the root to the node, and no other item which is more frequent than the item of the node. Note that depending on its prefix, a record might belong to the list of more than one nodes. For example, the record with $id = 1$ belongs to the lists of all its prefixes, i.e. it appears in the lists of the path a, $a \rightarrow c$ and $a \rightarrow c \rightarrow f$. Finally, there is a difference among the records that pertain "solely" to a node and the records that also pertain to its descendants. Observe the node c of the path $a \rightarrow c \rightarrow f$. The record with $id = 4$ is the record

$\{a, c, f\}$ that also belongs to the node f of the same path. On the contrary, the record with $id = 1$ refers exactly to the path $a \rightarrow c$. The distinction will be very useful later, for equality and superset queries.

The potentially very large number of descendants that a node might have and the fact that tries are unbalanced do not make the trie a good candidate for secondary memory storage. Therefore, we choose to use it as a main memory structure offering alternative access to the data, on top of the inverted file.

5.2.1.2 The HTI index

We know that the performance of the inverted file suffers, when very long inverted lists have to be processed. The issues involved in the processing of inverted files are (a) the IO cost of transferring the disk pages with the inverted lists to main memory and (b) the CPU cost of intersecting inverted lists of different items that participate in the same query set.

To counter this effect we propose the HTI index, which uses a relatively small main memory trie to offer additional access points to the inverted lists of the most frequent items (that also have the longest inverted lists).

The basic idea of the HTI-index is to split the vocabulary of the database into (a) a small set of frequent items I_{fr} and (b) a large set of infrequent items $I \setminus I_{fr}$. Then, a trie is used for the former, in order to speed up the access to the lists that pertain only to the combinations of frequent items, whereas the latter are treated as usually, through an inverted file.

The HTI-index, has three major components: a vocabulary, an access tree and a set of inverted lists. An HTI index is schematically depicted in Figure 5.2.

The *vocabulary.* Like inverted files, the HTI has a list of all the distinct items of the database, which offers access to the inverted lists. The items in the vocabulary are divided in two classes: (a) the *frequent* items I_{fr}, $I_{fr} \subseteq I$, whose vocabulary entries point to the access tree in main memory, and (b) the *infrequent* items, $I_{infr} = I \setminus I_{fr}$, whose vocabulary entries lead directly to their inverted lists in secondary storage, exactly like in inverted files. The vocabulary is kept as an array in main memory and together with the access tree root they comprise the initial access points to the inverted lists. The array is implemented as a hash table.

The *access tree.* The access tree is a trie structure that offers access points to blocks of records that share the same *access prefix paths* (*app*). The *app* of a record can easily be computed if we order its items according to the item frequency ordering of I. Then, we define as *access prefix path* the sequence prefix path whose items all lie in I_{fr} – i.e., the ordered sequence of the frequent items of the record. For example, the *app* of $\{a, f\}$ is $\{a\}$. We store the *app* of each record in the access tree, by putting the first and most frequent element as a direct child of the root (see

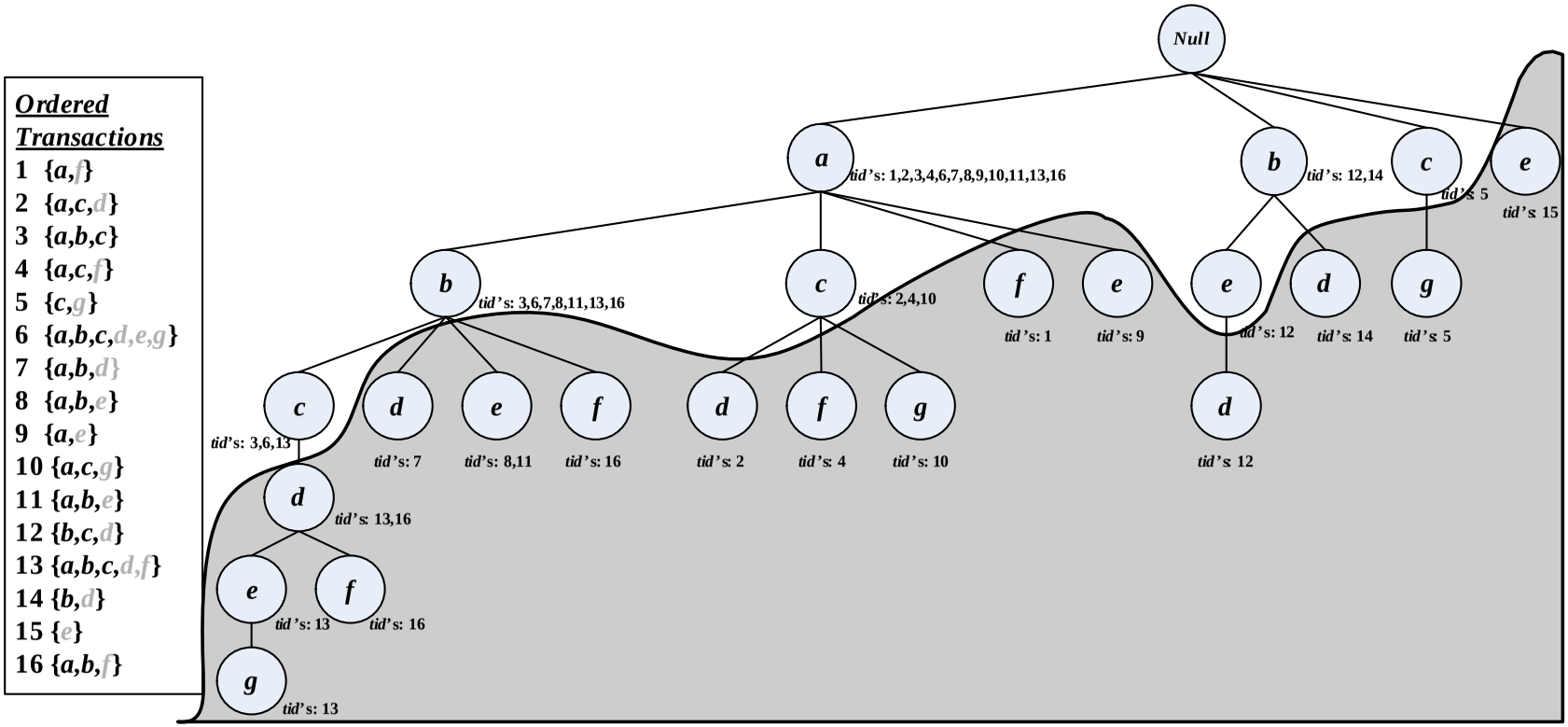


Figure 5.1: *An index based on a complete trie for the example of Figure 4.2*

also the next section for a detailed discussion on the creation of the access tree). The access tree has two kinds of nodes: (a) the *root*, which does not correspond to any item in I_{fr} and (b) *information nodes*, which are all the other nodes of the trie. Each such node holds the following information:

- A *label* indicating the item of I_{fr}, which corresponds to the node.
- A link to the inverted sublist of the records that contribute to the path from the root to the node. These are all the records whose prefix is the same with the path from the root to the current node.
- Navigational links to the children-nodes, the parent-node and to the rest of the nodes with the same *label*.

It is important to stress here that due to the vast volume of the full-fledged trie presented in the previous section, the access tree is a subset of it, concerning only its most frequent items I_{fr}. The vocabulary entries concerning these frequent items point to lists that comprise all the access tree nodes that are labelled with the respective item. In turn, these nodes point to the respective inverted lists, stored in secondary storage.

In Figure 5.2 we depict an example HTI index for the relation of Figure 4.2. We choose as frequent items $I_{fr} = a, b, c$ (having a frequency equal or greater than 8), and we create the access tree considering only them. Observe that in Figure 5.1, these were also the items with the longest record lists. The shaded area in Figure 5.1 concerns the infrequent items that were subsequently dropped from the access tree of Figure 5.2. Item a is more frequent than c, thus it precedes it in access tree paths. Assuming this I_{fr} set, all the records of Figure 4.2, contribute to three paths: $a \rightarrow b \rightarrow c$, $a \rightarrow c$, $b \rightarrow c$ and c. Observe, also, how the nodes with the same label are linked to each other.

The *inverted lists*. There are two cases for the inverted lists of the vocabulary items: (a) lists of non-frequent items and (b) lists of frequent items. Concerning the *non-frequent items*, their inverted lists are exactly the same as those of a regular inverted file (i.e., sorted lists containing the *id*'s of all the records that contain the respective item). The case of *frequent items* belonging to I_{fr}, on the other hand, involves inverted lists made up of many smaller sorted sublists, each of them corresponding to an access tree information node labelled with the respective item. To enhance the evaluation of equality and superset queries, we further divide the inverted sublists of the access tree to two parts as depicted in Figure 5.3, for the case of $a \rightarrow b$: (a) the $id's$ of the records whose *app* ends at this node; these are records $id's$ 7,8,11, and 16, (b) the $id's$ of rest of the records that contribute to the current node; these are $id's$ 3,6, and 13. In the beginning of each information node

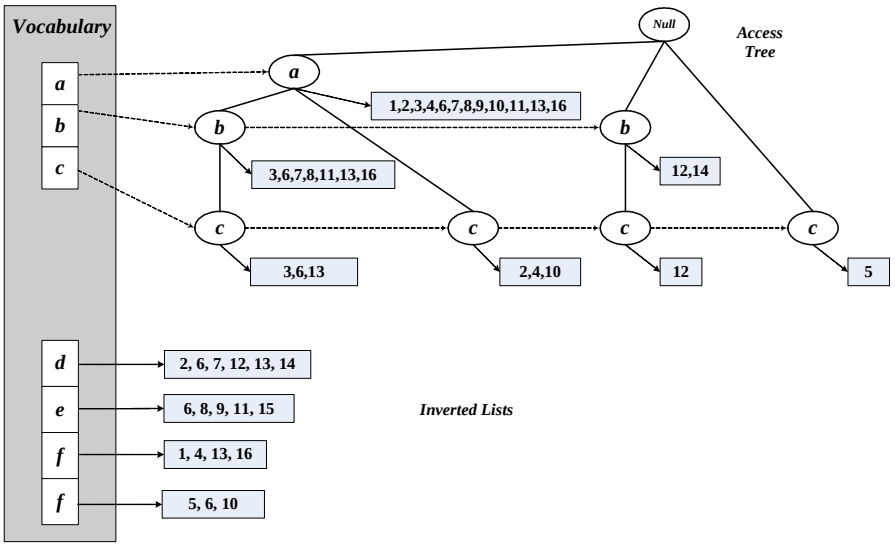


Figure 5.2: *HTI index for the relation of Figure 4.2.*

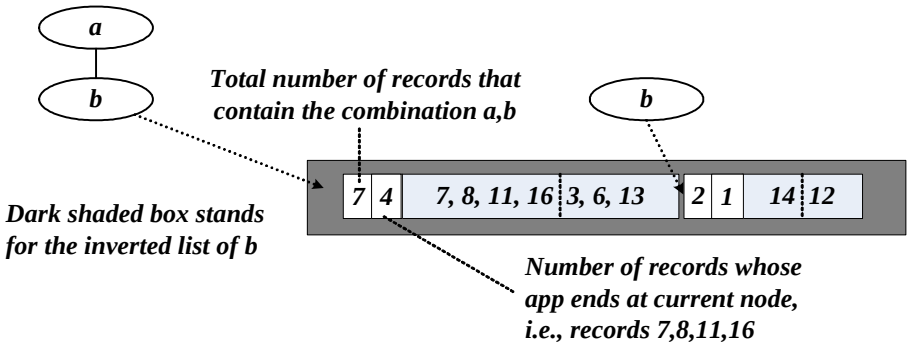


Figure 5.3: *The record list corresponding to the item b*

sublist, we store the number of records of case (a) alongside with the total number of the records that contribute to the current node, so that we can retrieve the right block from the disk each time.

Example. As shown in Figure 5.2, the access tree and the vocabulary are kept in the main memory, whereas the inverted lists reside at secondary storage. Records 1,3,4,6,7,8,9,10,11,13 and 16 contribute to the path a, thus they are stored at the inverted sublist of node a. Observe that a, being the most frequent item has exactly one inverted sublist, i.e. its inverted list comprises a single sublist, corresponding to its single appearance in the access tree. This is not necessarily the case for all the items, though. For example, the item b has two sublists. Seven of the records of a, (3,6,7,8,11,13, and 16), also contribute to the path $a \rightarrow b$, and they are stored in the first inverted sublist. At the same time, records 12 and 14 contribute to the path b and they are stored at the second sublist. For storage efficiency, the individual sublists of all the different nodes of b are stored contiguously, one after the other. The nodes of the access tree point to the offset of the inverted list where their

corresponding sublist begins. Note that in Figure 5.2 we depict only the *id*'s that are contained in the sublists and not the labels that mark each sublist for reasons of readability. A more complete representation of the sublist for the case of item a is depicted in Figure 5.3.

The rest of the items are indexed by an inverted file and the *id*'s of the records that include them are stored in the respective inverted lists. Note that, concerning the infrequent items d, e, f, g, their vocabulary entries point directly to the inverted lists in secondary storage without any interference with the access tree.

Compression and caching There is a question of how the HTI index compares to inverted files, when compression techniques are applied [SWYZ02, MZ96] or a cache equal to the access tree size is given to the inverted file. As far as the former is concerned, the HTI index is complementary to compression and not competitive to it. The inverted lists in the HTI can be compressed exactly like the inverted lists in a simple inverted file. Moreover, if the inverted lists become smaller, then we can reduce the size of the HTI by using a smaller threshold. Giving cache to the inverted lists on the other hand, may be a good solution for uniform distributions with large vocabularies. Still, the effectiveness of the cache is dependent on how big it is when compared with the total inverted file and it will be reduced as the size of the inverted file grows. On the contrary, the main memory requirements of the HTI index depend mostly on the size of the vocabulary, since duplicate or similar records do not affect its size and effectiveness. Thus, for small vocabularies and especially for skewed distributions, the HTI index is a better choice.

5.3 Query evaluation

In this section, we present the evaluation algorithms for the three types of queries that we are interested in: *subset*, *equality* and *superset*. The evaluation algorithms for all types of queries have two main stages: (a) evaluation in the access tree, and (b) evaluation in the inverted file. The evaluation in the access tree concerns the frequent items of the query set and the evaluation in the inverted file the rest of the items. The basic idea is that we use the access points to the inverted lists offered by the trie, to quickly trace the final or a candidate answer to the query. The benefit is quite significant since the access points are given for the largest lists, which correspond to the items of I_{fr}. This way we avoid expensive union or intersection operations between the lists indexed by the access tree, and instead we implicitly perform these operations in the tree itself.

Note that we are interested in identifying the id's of the records which belong to the answer. Once these id's have been identified, we can retrieve the respective records from the physical storage directly (see Section 2 for a related discussion). The

cost for retrieving the records is the same for all relevant indices, since neither the HTI, nor the inverted file, nor signature-based methods provide a storage scheme for the actual data. Thus, in our analysis we do not take this cost into account.

For all three cases of queries, we assume a query set of the form $qs = \{f_1, \ldots, f_k, i_{k+1}, \ldots, i_n\}$, where the first k items f_i concern the frequent items of the query set, belonging to the access tree, and the next $n - k$ items i_j are the infrequent items that are only indexed by the inverted file.

In the following, we detail the evaluation techniques for each type of queries.

5.3.1 Subset queries

Subset queries are the most common queries executed against record and text collections and most broadly studied in research literature. Furthermore, the evaluation of many query classes, including ranking ones, partially resolves to the evaluation of subset queries.

The main idea around evaluating subset queries is that the records that contain the *app* part of the *qs* can easily be identified by using the access tree, without merging the respective inverted lists. This is efficiently done by tracing all the appearances of the last element of the *app*, f_k (which is also the least frequent in *app*), and then identifying which paths from the *root* to the f_k nodes contain the *app* of the *qs*. These paths possibly contain other frequent items too, but they necessarily contain the *app* of the query set. We call the set of the retrieved record id's $candidateIds$. Possibly, apart from the frequent items, there are also infrequent items in the query set. The only way to access these infrequent items $i_{k+1}, \ldots, i_n$ is through the inverted file. Therefore, to compute the final query answer we must find the intersection of the lists of record *id*'s that correspond to the infrequent items $i_{k+1}, \ldots, i_n$ with the list of the already retrieved $candidateIds$. Any record *id* that belongs to this result contains both the frequent items of the *app* and the infrequent items $i_{k+1}, \ldots, i_n$. The algorithm in pseudo-code is depicted in Figure 5.4.

Assume for example that the user asks for all records that contain the $\{f, c, a\}$ items from the relation D depicted in Figure 4.2. If we evaluate the query against the inverted file, depicted in Figure 4.3, we would have to perform a merge-join of the inverted lists of all the items in the query set. That would require six disk page accesses and we would only have one answer, that is $t.id = 4$. If, instead, we evaluate the query against the HTI index, the disk pages accesses are much fewer. First, we have to identify the *app* of the *qs* which is $a \rightarrow c$. Then, we must trace all the c nodes of the access tree and identify the paths from *root* to c, which contain the rest of the items of *app*, i.e., f. This results in only one path: $root \rightarrow a \rightarrow c$. Now we can directly retrieve the records that contain a and c, which are 2,4, and

Algorithm SubsetQueries
Input: An HTI index H over a dataset D, a query set $qs = \{f_1, \ldots, f_k, i_{k+1}, \ldots, i_n\}$ and a query $Q=\{t \mid qs \subseteq t.s\}$.
Output: the $t.id$'s of the records that contain qs
Method:

1. Determine the $app = \{f_1, \ldots, f_k\}$ of the query set.
2. If app is not empty use subsetTrie(app) to retrieve the $candidateIds$ from the trie.
3. If $\{i_{k+1}, \ldots, i_n\}$ is not empty in the query set:
4. $result$=merge-join the $candidateIds$ with the inverted lists of $\{i_{k+1}, \ldots, i_n\}$
5. else
6. $result$=$candidateIds$
7. return $result$

Function subsetTrie(app)
Input: An HTI index H over a dataset D, the app of the qs
Output: The $candidateIds$, i.e. the $t.id$'s of the records that contain the items of app
Method:

1. Let c be the last item (least frequent) of app
2. For every appearance of c in the trie
3. if every item $f_i \in app$ appears in the path from the $root$ to the current node.
4. add the $t.ids$ of the inverted sublists of the current node to the $candidateIds$
5. return $candidateIds$

Figure 5.4: *Algorithm for determining subset queries*

10 by performing only one page access. Subsequently we can merge-join $\{2, 4, 10\}$ with the inverted list of a to retrieve the final answer. The total disk page accesses we encounter in this case is two. In general, if the inverted lists of the items of the qs (ordered by frequency) cover $l_1, \ldots, l_n$ disk pages, the worst case evaluation will require $l_1 + \cdots + l_n$ disk page accesses. This holds for both the inverted file and HTI index, but as experiments in Section 5.5 show, the average cases clearly favor the HTI index. The benefit from using the access tree comes from the fact that we avoid performing intersections between the largest inverted lists. This benefit can potentially be very significant, especially if the frequent items are not correlated. Moreover, the larger the inverted lists are and the greater the skewness of the items distribution is, the greater benefit we gain from using the access tree.

Some more technical notes should also be made for the algorithm of Figure 5.4. Whereas the simplified form of the algorithm, implies that we use the access tree to actually retrieve the $t.ids$ from the disk and put them in the $candidateIds$ this is not the most effective implementation in most cases. Instead, we return the links to the inverted sublists in the disk, which are then merged-joined with the inverted lists of the $\{i_{k+1}, \ldots, i_n\}$ items. Furthermore, the merge-join is performed by starting from

the less frequent item, thus it is not always necessary to use the access tree. In some cases, we can quickly decide that there is no solution, by intersecting the smaller inverted lists, and avoid any further computation. Practically, the algorithm first traverses the access tree and decides if there is a solution for the *app* items, and how many disk page accesses it will need to retrieve them. Depending on how many disk page accesses it will need, the algorithm decides the order of the merge-joins, i.e., whether it will start from the trie or the inverted file.

5.3.2 Equality queries

Employing the HTI index for equality queries leads to very efficient evaluations. For each query, only one path of the access tree has to be identified. This is the path, which is identical to the *app* of the query set. Assuming that nodes are organized in some efficient data structure, like hash arrays, the evaluation on the trie can be done in time $O(|app|)$, that is proportional to the *app* of the query set. After identifying the single inverted sublist that possibly satisfies the query, it has to be intersected with the inverted lists of the non-frequent items. In the process of the merge-join, the records are filtered according to their length, which must be equal to $|qs|$. The pseudocode of the evaluation algorithm is depicted in Figure 5.5. The worst case in terms of page accesses is again the same as for subset queries. Still, experiments show that whereas evaluating equality queries in the inverted file requires as many disk page accesses as the respective subset queries did, the results with HTI index are a lot better in this case.

5.3.3 Superset queries

Superset queries are by far the most expensive queries we study. In a sense, a superset query is equivalent to $2^{|qs|}$ equality queries, for all its subsets. The evaluation algorithms, even those that work only in the inverted file, require significantly less disk page accesses than $2^{|qs|}$ equality queries, but still the number is high. If the inverted lists of the items of the qs (ordered by frequency) need $l_1, \ldots, l_n$ disk pages respectively, evaluating a superset query solely in the inverted file, with the algorithm presented in Figure 5.6, requires in the worst case $l_1 + 2l_2 + \cdots + nl_n$ disk page accesses.

As in the case of equality, the access tree can drastically boost the efficiency of the query evaluation. The basic idea is to find all the paths in the trie, which are solely constructed by items from the *app* of the query. Then we can safely add to $candidateIds$, the ids of all the records that end in any node of these paths. For these records we know that they do not contain any other item of I_{fr}, except from $f_1, \ldots, f_k$. If the qs has non frequent items too, then we have to check in the inverted

Algorithm EqualityQueries
Input: An HTI index H over a dataset D, a query set $qs = \{f_1, \ldots, f_k, i_{k+1}, \ldots, i_n\}$ and a query Q=$\{t \mid qs = t.s\}$.
Output: the $t.id$'s of the records that are equal to qs
Method:

1. Determine the $app = \{f_1, \ldots, f_k\}$ of the query set.
2. If app is not empty use equalityTrie(app) to retrieve the $candidateIds$ from the trie.
3. If $\{i_{k+1}, \ldots, i_n\}$ is not empty in the query set:
4. $result$=merge-join the $candidateIds$ with the inverted lists of $\{i_{k+1}, \ldots, i_n\}$
5. else
6. $result$=$candidateIds$
7. return $result$

Function equalityTrie(app)
Input: An HTI index H over a dataset D, the app of the qs
Output: The $candidateIds$, i.e. the $t.id$'s of the records that contain exactly the items of app
Method:

1. Let c be the last item (least frequent) of app
2. For every appearance of c in the trie
3. if every item $f_i \in app$ and only these items, appear in the path from the $root$ to the current node.
4. add the $t.ids$ of the inverted sublists of the current node to the $candidateIds$
5. return $candidateIds$

Figure 5.5: *Algorithm for evaluating queries*

file if the remaining items of the records of $candidateIds$ contain only items from $i_{k+1}, \ldots, i_n$. If the qs does not contain any other items we filter the $candidateIds$ using their length and the length of the path that lead to them, as pruning criteria. If their length is greater than their app, which can be inferred from the trie without examining the record itself, the record is dropped, since it must have more items that are not contained in qs. The algorithm for evaluating the superset query is presented in Figure 5.6.

The reduction of the disk pages accessed, when using the HTI index for superset queries, is not only attributed to the access points offered by the trie. It is also a result of the possibility of identifying exactly the records whose app ends at the access tree nodes, as opposed to the rest of the records within the same inverted sublist.

Algorithm SupersetQueries
Input: An HTI index H over a dataset D, a query set $qs = \{f_1, \ldots, f_k, i_{k+1}, \ldots, i_n\}$ and a query $Q=\{t \mid qs \supseteq t.s\}$.
Output: the $t.id$'s of the records that where $t.s \subset qs$
Method:

1. Determine the $app = \{f_1, \ldots, f_k\}$ of the query set.
2. If app is not empty use supersetTrie(app,$root$) to retrieve the $candidateIds$ from the trie.
3. Let $il_1 \ldots il_m$ be the inverted lists of all the non frequent items of the qs and the $candidateIds$, ordered according to the number of memory pages

4. for (i=1 ; $i \leq n$; i++)
5. for each entry t of il_i
6. $unmatched$=$t.length - 1$
7. if ($unmatched == 0$) add t to $result$ and break
8. for ($j = i + 1$; $j \leq n$; j++)
9. if ($unmatched > n - j$) break
10. if ($unmatched$==0) add t to $result$ and break
11. scan forward il_j
12. if t found in il_j $unmatched = unmatched - 1$
13. return $result$

Function supersetTrie(app,$currentNode$)
Input: An HTI index H over a dataset D, the app of the qs, the $root$ of the trie as $currentNode$
Output: The $candidateIds$, i.e. the $t.id$'s of the records whose items are contained in app
Method:

1. while (app not empty)
2. $newCNode$=pop(app)
3. if $newCNode$ is child of $currentNode$
4. add the inverted sublist of $newCNode$ to $candidateIDs$
5. supersetTrie(app,$newCNode$)
6. return $candidateIds$

Figure 5.6: *Algorithm for determining superset queries*

5.4 Creation and Maintenance

5.4.1 Construction of the HTI index

The construction of the HTI index differs from the construction of the inverted file in two points: (a) the additional construction of the access tree and (b) the detection of the frequent items of the database, which are indexed by the access tree. The former point does not present any serious problem, since the access tree is kept in memory; we present the related insert/delete algorithms in the following. The latter point introduces a more substantial difficulty in the creation; we need to read all the input once just to decide which items are frequent, and then we

need a second, separate, scan to find all their combinations that must be traced in the access tree. Thus, the two-pass algorithm we presented in Section 4.5, will be a three-pass algorithm and merge algorithm will require two passes. A way around this overhead, is to use *sampling* in order to decide which items are frequent enough to be indexed in the access tree. Sampling might not lead to the optimal item choice, but the effect is limited as we show in experiments we performed over distorted orderings. In practice, this is not a significant problem, since in several application fields, like web and supermarket logs, the vocabulary is known a priori and the most frequent items, can be easily estimated. If we know the I_{fr}, then any new combination that we meet in the processing of the raw data is handled exactly as a new word is handled in text documents; a new entry is inserted in the access tree and in the inverted sublists of each affected node. The algorithms for inserting and deleting a single record from the HTI are presented in the following.

Insertions. For the case of the insertion of a record $t = \{o_1, \ldots, o_k, o_{k+1}, \ldots, o_n\}$ of length n, having $app = \{o_1, \ldots, o_k\},\ k \leq n$, to the database D we have to take the following actions (the algorithm in pseudo-code is presented in Figure 5.7):

1. For all the infrequent items of t that do *not* appear in *app*, the record *id* must be added in their inverted list, in secondary storage.

2. The *app* path must be added to the access tree (if not already there) and the record id to the respective record lists. Possibly, some prefix of *app*, say $\{o_1 \rightarrow \cdots \rightarrow o_e\},\ e \leq k$ already exists in the access tree as a result of a previous insertion. For all the nodes belonging to the maximal such part of the *app*, we simply add the new record t to their inverted list. The rest part of the *app*, $o_{e+1} \rightarrow \cdots \rightarrow f_k$ that has not been mapped in the trie as part of the proper path is added as a child to the node that corresponds to item o_e.

If the record does not have any common prefix with a previously inserted record, the whole *app* of the record forms a new path in the trie, starting from the root. If on the other hand, its *app* is the same with the *app* of a previously inserted record, its insertion does not cause the insertion of any new nodes in the access tree.

Deletions. For the case of deletions the steps are practically the same. For the infrequent items that are indexed only in the inverted file but not in the trie, we simply remove their record *id* from the record lists. For the frequent items of the record, we locate the path of the trie that corresponds to the *app* of the record. Then, we remove the record *id* from all the inverted lists of all the nodes in this path. In the case that the inverted list of a node becomes empty, then we remove the node from the access tree. By the definition of the trie, if a non-leaf node has an empty inverted list, then all its children obligatorily have an empty inverted list,

```
Function boolean insertPath(app,t.id){
 1. currentNode=root
 2. while (app not empty) {
 3.     firstLabel=pop(app)
 4.     if (exists child c of currentNode
                        with label=firstLabel){
 5.         add t.id to the inverted sublist of c
 6.     } else {
 7.         add a new node with label firstLabel
                        as a child of currentNode{
 8.     }
 9.     currentNode= c
10. }
```

Figure 5.7: *Pseudo-code for the* `insertPath` *function that inserts into the trie the record* t *that has* $t.id$ *as an id and app as its access prefix path.*

too. Therefore, the whole path from the non-leaf node to its descendant leaves is directly removed.

The algorithm for deleting a record from the trie tree is presented in Figure 5.8.

```
Function boolean deletePath(app,t.id){
 1. currentNode=root
 2. while (app not empty) {
 3.     firstLabel=pop(app)
 4.     find the child c of currentNode
                        with label=firstLabel
 5.     remove t.id from the inverted sublist of c
 6.     if the list is now empty{
 7.         remove the sub-tree of c
 8.         break
 9.     }
10.     currentNode= c
11. }
```

Figure 5.8: *Pseudo-code for the* `deletePath` *function that deletes the record* t *that has* $t.id$ *as an id and app as its access prefix path.*

5.4.2 Updates in the HTI index

Updates in a transactional context are usually insertions, with deletions being rate in most application areas. When a ΔD set of records is inserted or deleted from a database D, there are basically two issues that have to be addressed:

1. Changes in the items of I_{fr}. Some items that were not frequent might now be frequent, thus they must be included in I_{fr} and some others must be removed from it.

2. There might be changes in the order of items of I_{fr}, thus the access tree must be restructured to reflect these changes.

3. The new records must be accommodated in the inverted lists.

If, there is no case in the items of I_{fr} and in their order and we only have to address the accommodation of the new records in the inverted lists, the solution is simple; we handle updates using the same techniques used for the inverted file. New postings are kept in inverted lists in main memory until they are merged with the inverted lists that reside in secondary storage. The merging can be performed following any of the available strategies (rebuild, merge, in-place) depending of the requirements of each application. Since, the access tree resides in main memory any additions in the HTI can be easily performed when we process the new data. Addressing the first and the second issue is rather more complicated and it is explained in Section 5.4.2.1.

5.4.2.1 Updates to the frequencies of the underlying data

Updates in the data can incur changes in the items of I_{fr} or just changes in their order. In most application areas that involve recorded data like retail store records, the relative frequencies of the items change slowly or remain stable. In any case, we keep the most frequent items in the access tree in order to improve query performance, and we order them internally by frequency, to keep the size of the access tree small, as reported in [HPYM00]. The proposed query evaluation algorithms are correct for any ordering and their performance does not deteriorate fast if the actual frequencies of the items do not follow closely the ordering used for them. Small changes in the item frequencies, even if they cause changes in the ordering of items, do not incur significant changes in the size of the access tree, or in the evaluation of containment queries over them. This is verified by the experiments we performed on data with wrong orderings for the items of I. To see how the size of the access tree and the performance of the queries is affected, when the actual frequencies of the data diverge from the ordering we have adopted, we reordered the items of a dataset taking into account distorted frequencies. We created 5 different HTI's over a dataset of 1M records, by keeping only the top 1% of the most frequent items in the access tree, each time assuming frequencies that diverge from the actual ones by 10%, 20%, 30%, 40% and 50%, on average[1]. The results are depicted in Figure 5.9. The basic conclusions that can be drawn are two: (a) that the performance of the HTI deteriorates as the actual frequencies diverge more and more from the assumed ones, but not very fast, and (b) that, at the same time, the size of the

[1]the maximum difference of the distorted frequencies, from the actual ones is 20%, 40%, 60%, 80% and 100% respectively

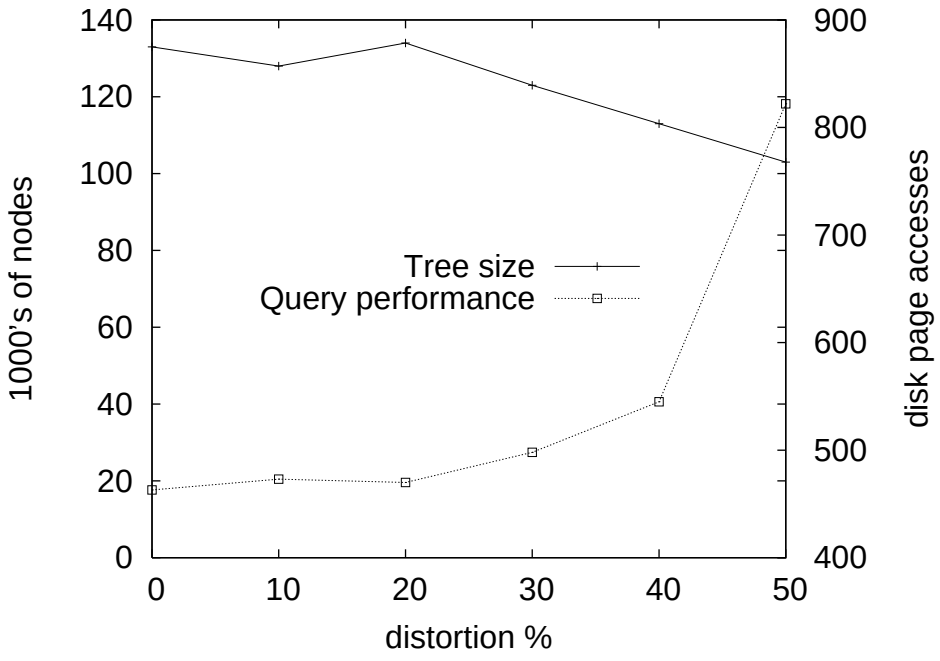


Figure 5.9: *Changes in the access tree size and query performance as the real frequencies of the data change*

access tree decreases slightly. The performance of the HTI deteriorates, since the wrong frequencies lead to the exclusion of some frequent items from the access tree. The same fact, that some less frequent items are kept in the access tree instead of the most frequent ones, is the cause of the decrease in the size of the access tree; this is a consequence of the fact that the access tree represents significantly less item appearances from the database. There is only one exception, when the ordering of the items is based on frequencies that diverge by 20% from the real ones. In this case, the size of the access tree is increased. This is attributed to a different mechanism that works in parallel; some frequent items appear in wrong positions in the access tree, thus less paths share the same prefixes. For example if item f precedes item a, but item a is in reality more frequent that item f, this will result in more nodes for representing the appearances of both a and f, than we would have if the ordering was correct. Nevertheless, it should be noted, that the items in the test dataset follow a Zipfian distribution of order 1. This means that the differences in their frequencies of appearance are quite significant and randomly distorting them by up to 20% or 40% does not cause any substantial changes in the ordering of the most frequent items.

Even if the changes are significant we can still avoid rebuilding the HTI from scratch. For the items that are no longer frequent, we have to delete all their appearances from the access tree and merge their inverted sublists to one inverted list. For the items that become now frequent, we have to delete and re-insert all the

records that contain them. We can avoid scanning the whole database, since we can easily detect the affected records from the respective inverted list. For changes in the ordering of I_{fr} we have to restructure the affected paths of the access tree and appropriately handle their inverted sublists. The basic steps for updating the HTI index are the following:

Assuming that the current ranking for I_{fr} is $o_1, \ldots, o_n$, and the new ranking for $I_{fr}^{'}$ is $o_1^{'}, \ldots, o_n^{'}$ the steps we must follow for restructuring the HTI are the following:

1. First we delete all the nodes of the access tree that correspond to items that are no longer in I_{fr}.

2. Then we delete all transactions that contain any of the new items in I_{fr} from the access tree.

3. Starting from the most frequent item $o_i^{'}$ of $I_{fr}^{'}$, for which it does not hold $o_i^{'} = o_i$ we apply $moveItem(o_i^{'}, i^{'})$. We update the current ranking, and repeat for all the items of $I_{fr}^{'}$, which still do not appear in the right position.

4. We add to the access tree all the transactions that contain new items of $I_{fr}^{'}$.

Function boolean moveItem(*item*,*newRank*){
1. For each appearance *itemNode* of *item* {
2. For each node i in the path from the *itemNode* to the *root*
3. if (i.currentRank $\leq$ *item*.newRank)
4. add the i.list to the *itemNode*.list by merge
4. add the *itemNode* under the first i whose rank is greater than *newRank*
5. delete *itemNode* from its old position and link its descendants to its parent
6. }
7. merge any sibling appearances of *itemNode*
8. update the *currentRank* of all items that were ranked between *oldRank* and *newRank* by subtracting 1

Figure 5.10: *Pseudo-code for the* `move` *function that moves a node from each current position s to a new position d higher in the same path.*

5.5 Experimental Evaluation for the HTI index

Evaluation metrics. We evaluate the HTI index by considering two main factors: (a) the benefit it provides to query evaluation, compared to regular inverted files and (b) the main memory requirements it imposes. We evaluate the benefit to query evaluation by counting disk page accesses as the dominant factor of the problem. We show how main memory requirements are affected for the different D parameters by providing the number of access tree nodes.

Data. We evaluated the HTI using two real datasets from UCI KDD repository [HB99]. Both datasets are web logs that trace the areas visited in single users' session in a specific portal. The containments queries have intuitive meanings in all cases, e.g. (superset) "Which users limited their visit in the portal in the main and downloads sections?". The first dataset, denoted as *msweb*, is a one-week log tracing the virtual areas that users visited in the web portal (`www.microsoft.com`). There are 32k records and the vocabulary of the dataset contains 294 distinct items (areas). The distribution of the items in the records is skewed and the average size of the record is 3. To be able to draw informative conclusions we replicated the dataset 10 times. This replication is meaningful, since it simply represents a 10-week log from the web portal. The second dataset, denoted *msnbc* is again a log of users' behavior of another web portal, the `msnbc.com`. The vocabulary here is very limited, comprising only 17 distinct items and unlike the previous one, the distribution of the items is relatively uniform. The average size of the record is 5.7.

Experimental setup. We implemented both methods in C, on a Linux platform (Suse 9.3) and compiled it with gcc version 3.3.4. Our experiments were performed on an AMD Sempron 2800+ with 2G of main memory. The disk page accesses were directly counted by the program, by tracing how many of the $4k$ arrays were accessed.

5.5.1 Performance of the HTI index

5.5.1.1 Real data

To measure the benefit on query evaluation provided by the HTI on real data, we evaluated subset, equality, and superset queries against the inverted file and the HTI index. For the case of the HTI index we varied the threshold, i.e., the percentage of items that comprise the I_{fr}. The results are depicted in Figures 5.11 and 5.12. For the case of *msweb* data, which are skewed but they have larger vocabulary than *msnbc* data, we used as thresholds 5%, 20%, 40%. The size of the access tree that must be kept in main memory is small in all cases, with the biggest being around 350k, for threshold 40%. For the case of *msnbc* data, where the vocabulary is very small, we used the thresholds 20%, 60% and 100%. The largest access tree in this case is around 200k, for threshold 100%. Note that for a threshold of 100%, all items of I are indexed by the access tree, thus for all types of queries no false positives are retrieved from the disk (we can infer the length of a transaction by the length of the access tree path if all items are indexed by the access tree).

As we can see the HTI index outperforms the inverted file in all cases. Moreover, it scales a lot better as the size of the query grows. For the larger queries, the performance of HTI (with a suitable threshold) is at least an order of magnitude

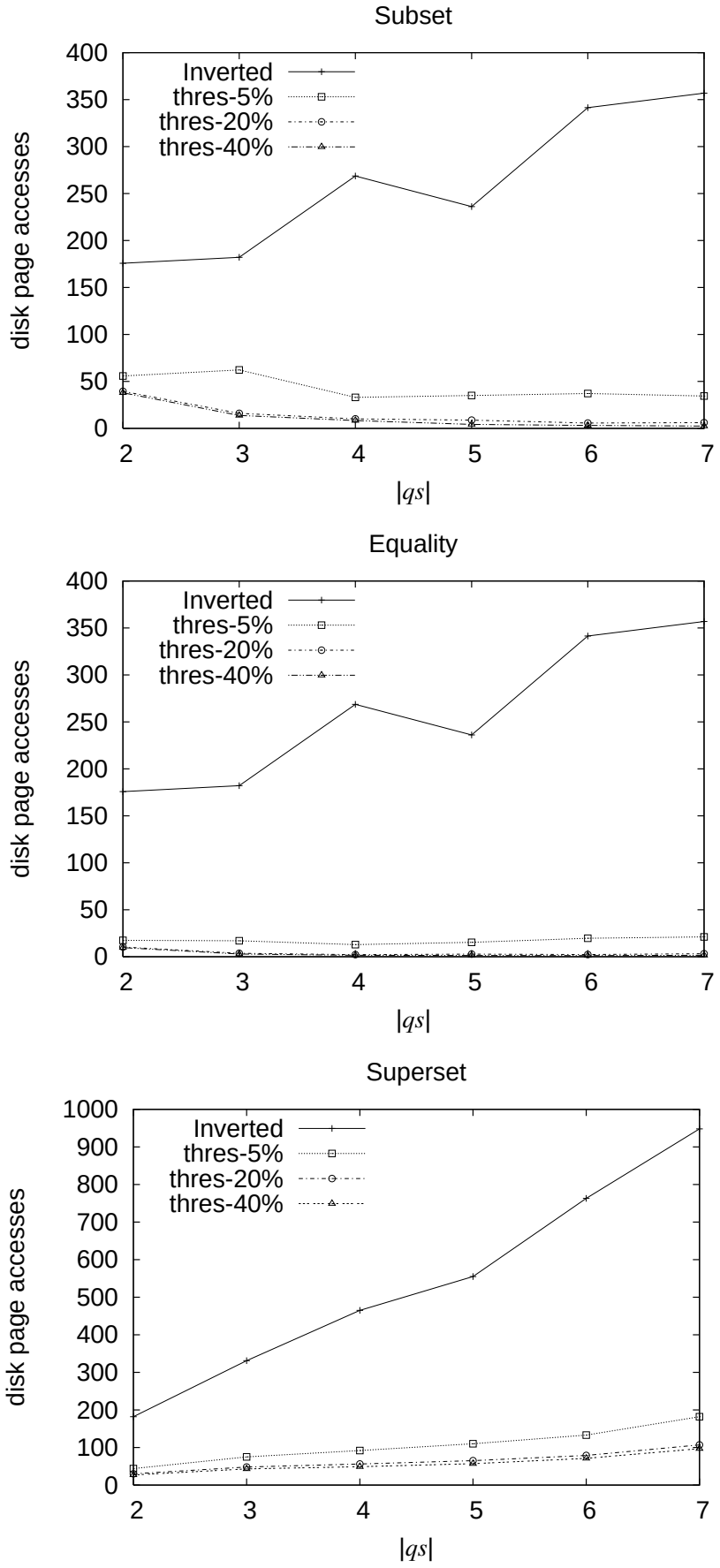


Figure 5.11: *Average performance of queries on msweb data*

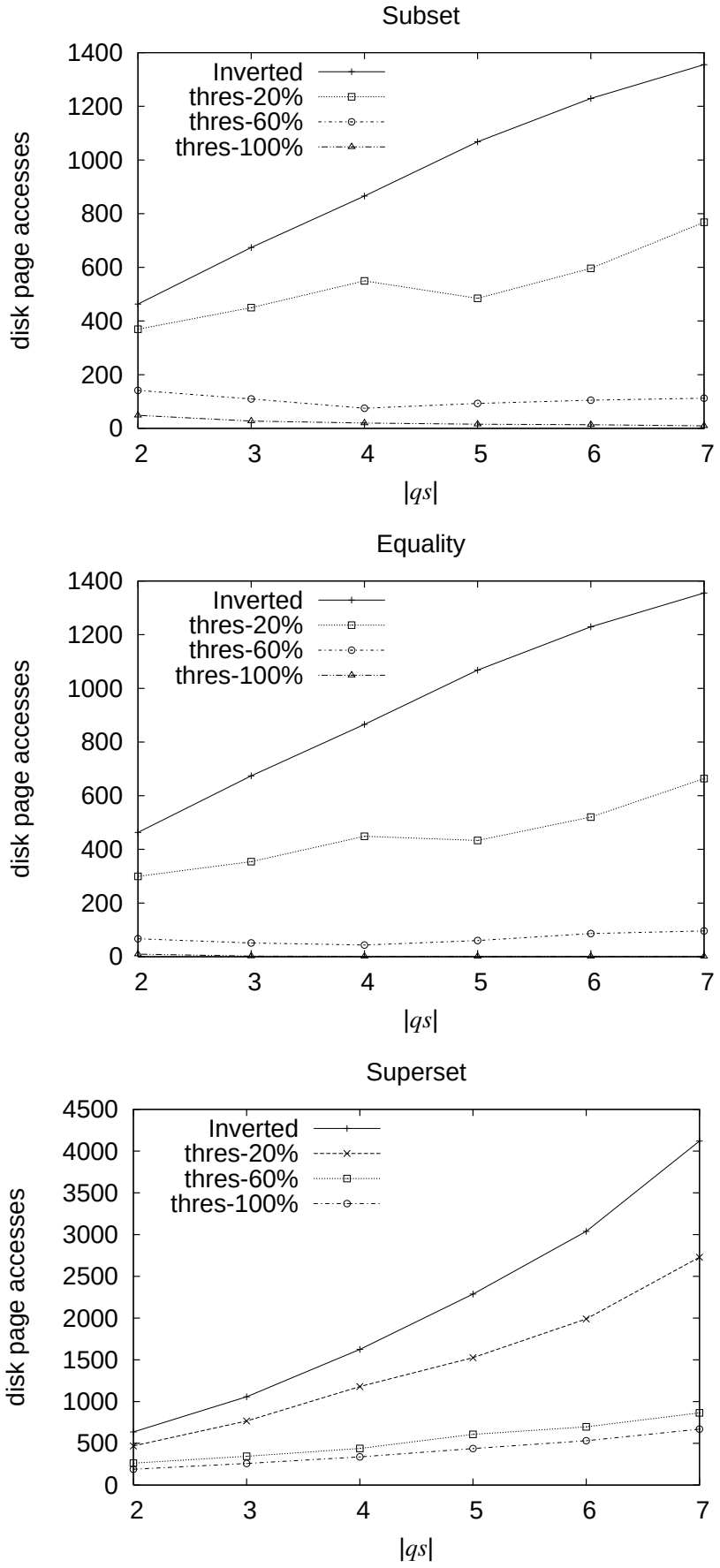


Figure 5.12: *Average performance of queries on msnbc data*

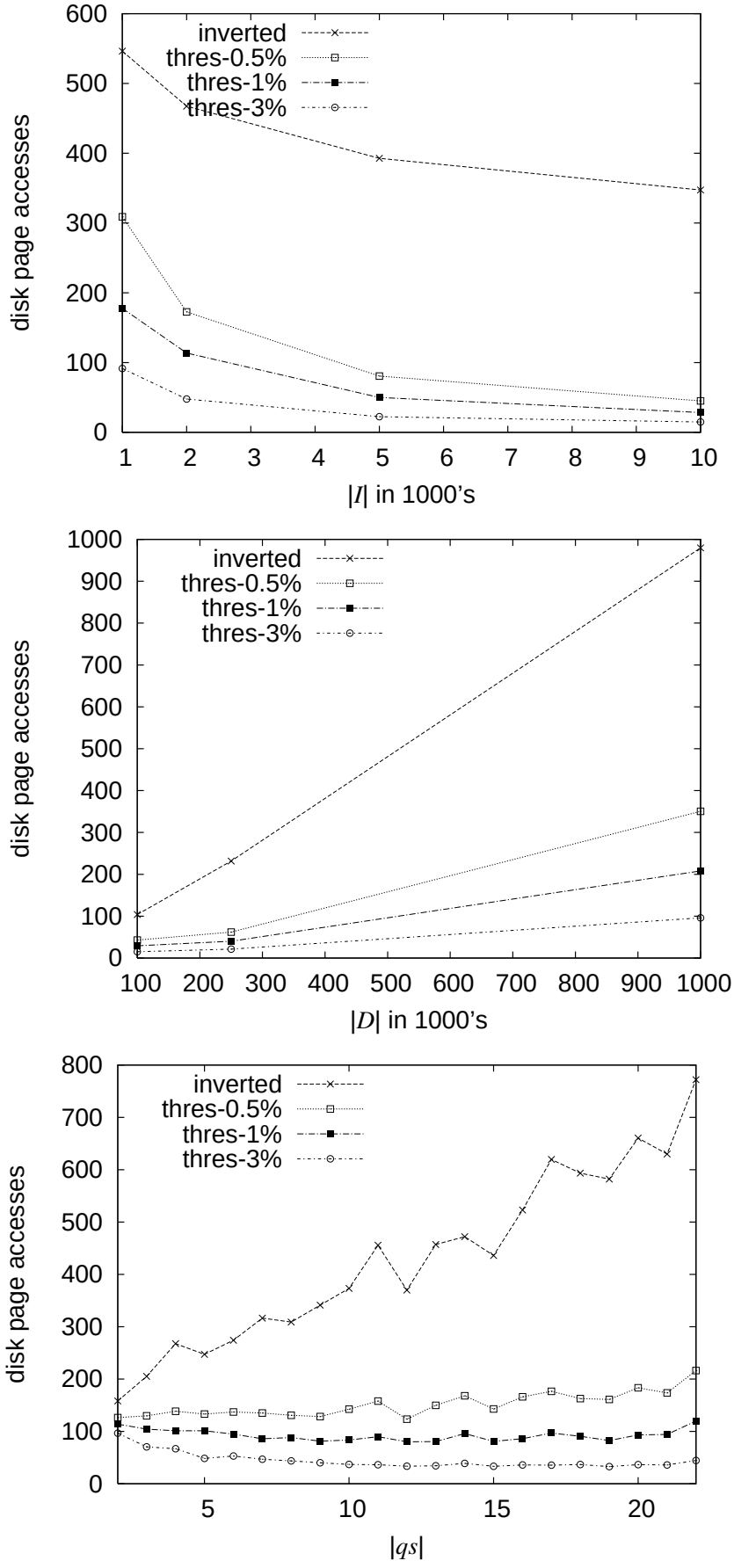


Figure 5.13: *Average performance of subset queries*

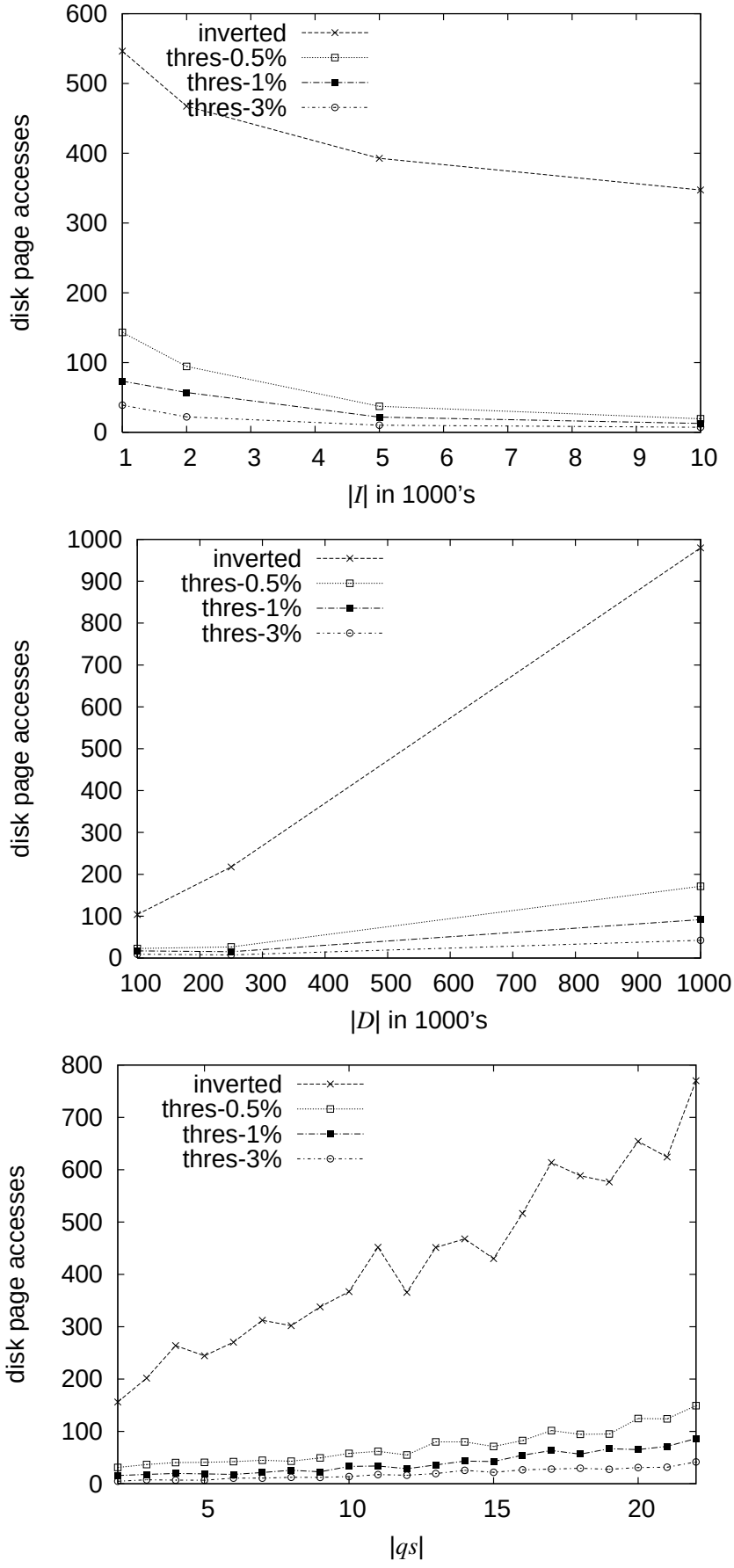


Figure 5.14: *Average performance of equality queries*

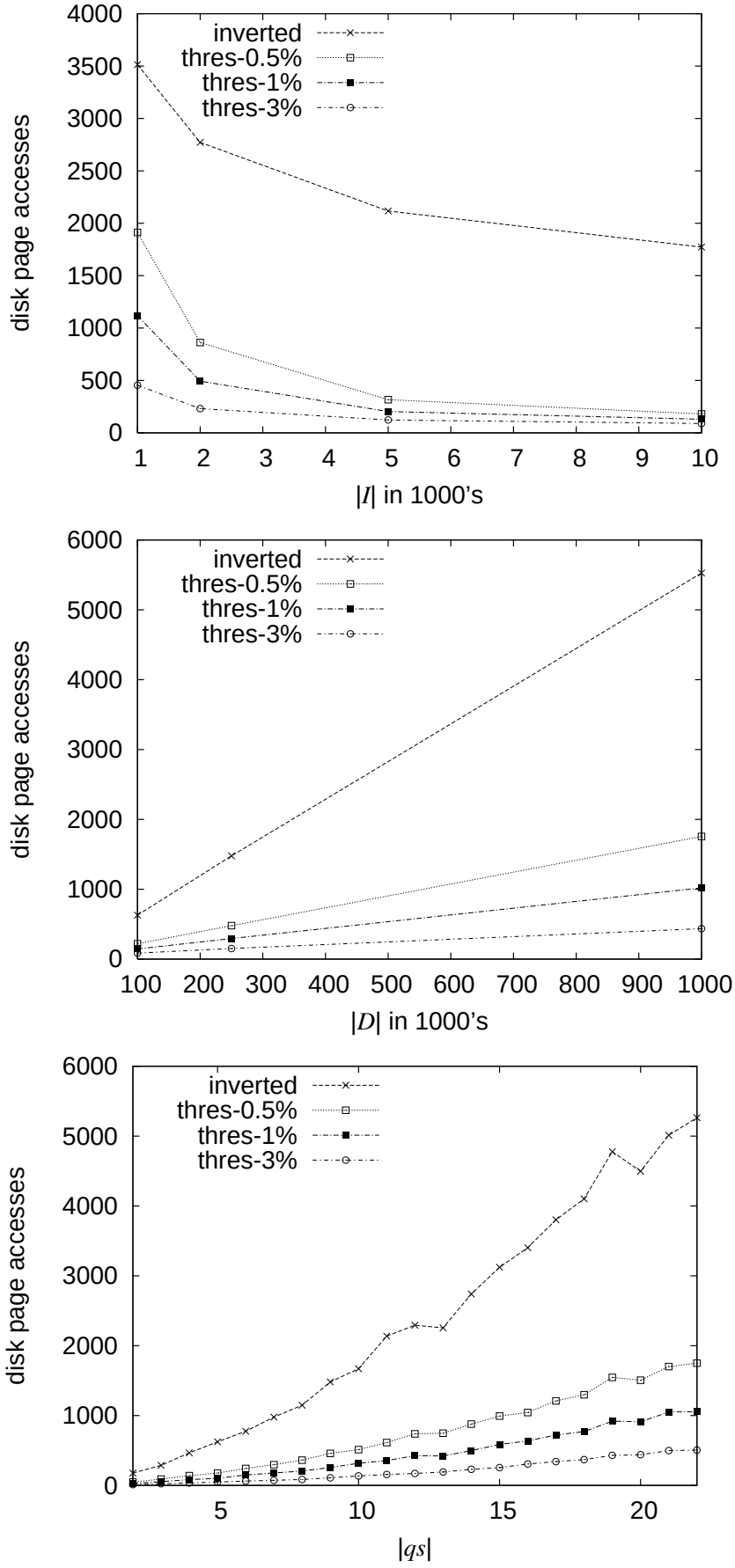


Figure 5.15: *Average performance of superset queries*

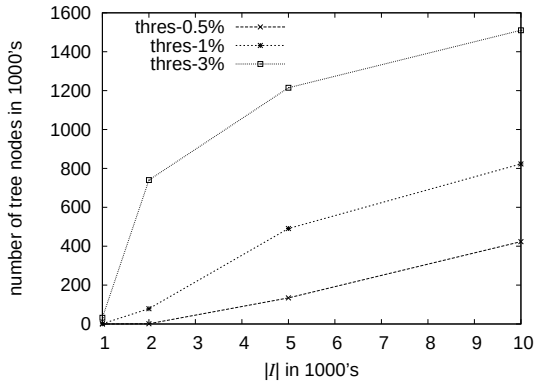


Figure 5.16: *Effect of* $|I|$

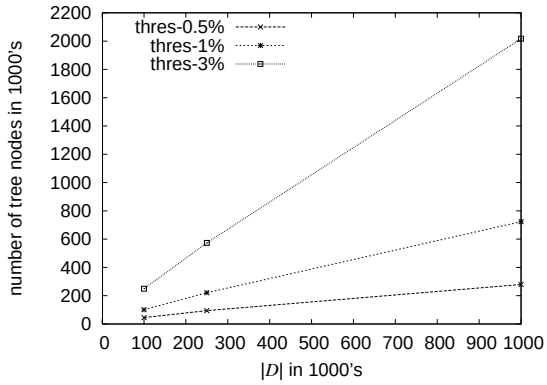


Figure 5.17: *Effect of* $|D|$

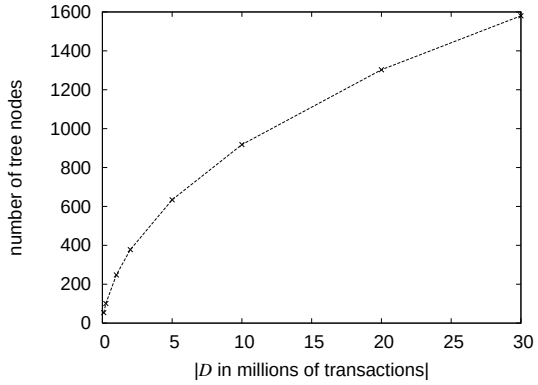


Figure 5.18: $|I| = 5k$, *0.5%*

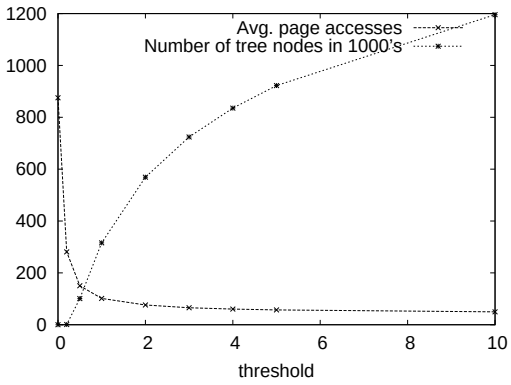


Figure 5.19: *Effect of* k

better for all types of queries.

5.5.1.2 Synthetic data

By using synthetic data we are able to trace the impact of the vocabulary I, the size of the dataset D and the size of the query set qs on the HTI index. In the following we investigate how each of the query types we introduced is affected by these factors.

Subset. In Figure 6.9 we see how the inverted file and the HTI index perform for subset queries. We compare three versions of HTI index with the inverted file, each time varying the threshold. Consider the first variant of the HTI index with a I_{fr} of only the top 0.5% of the total items. In all three experiments of Figure 6.9, we count the average number of page accesses performed by all our queries on all our datasets as a function of (a) the size of the vocabulary, I (left); (b) the size of the underlying database D (center), and (c) the number of items belonging to the query set qs (right). In all three cases, results are given for the average value of all parameters that do not appear in each figure. Thus, when varying $|D|$, we present the average of the results for all $|I|$ and $|qs|$, when we vary $|I|$ we present the average of the results for all $|D|$ and $|qs|$, and when we vary $|qs|$ we present the average of the results for all $|D|$ and $|I|$. An overview of the individual results showed that

they obey the general trend.

In all cases, the HTI index outperforms the inverted file by a significant factor. It is important to note that the HTI seems to scale a lot better for large databases and large queries. Whereas in the average case the increase of $|D|$ seems to have a linear impact on the disk page accesses for both methods, the gradient of the HTI index performance is significantly smaller. The larger the threshold is, the smaller the disk page access increase is. Furthermore, the increase of the $|qs|$ has diverting impact on the performance of the inverted file and the HTI index. In the former case it is followed by a proportional increase in disk page accesses, whereas in the latter case the required number of page accesses is reduced. This is due to the fact that when dealing with large queries, the chance of having more items from I_{fr} is greater, thus the chance of performing a more effective pruning in the accesses tree is greater.

The increase of the vocabulary size seems beneficial both for the HTI index and the inverted file, but as we show in the experiments for the HTI size, it significantly augments the memory requirements for the access tree.

Equality. Equality queries favor the HTI-index even more. In Figure 6.10 we assess the number of disk page accesses for equality queries as a function of (a) the vocabulary size, $|I|$ (left), (b) the size of the underlying database, $|D|$ (center) and the number of items of the query set, $|qs|$ (right). The evaluation in the inverted file requires exactly the same disk page accesses for equality queries, as it did for subset queries. On the other hand, evaluating equality queries in the HTI requires less than half of the disk pages accesses it did for the respective subset ones. This effect is even greater for queries with low cardinality qs. The main reason that makes equality queries behave better with the HTI index is that each query requires retrieving one inverted list from the access tree at most.

Superset. As it can be inferred from Figure 6.11 in superset queries the HTI-index clearly outperforms the inverted file index. The inverted file performs very poorly, since it requires multiple scans of many lists. Note that the disk page accesses performed in the evaluation of the superset queries surpass the disk page accesses needed by subset and equality queries by almost an order of magnitude.

5.5.2 Memory requirements of the HTI index

The size of the access tree of the HTI index for the real datasets we used is very small; for the case of the *msweb* data it has only 1857 nodes (around $33kb$) for a threshold of 5%, and in the worst case (threshold 40%) it has 20569 nodes (around $369kb$). For the case of *msnbc* data, it has only 7 nodes for a threshold of 20% and in the worst case (threshold 100%) it has 11575 nodes ($206kb$). The size of the

access tree is important, since it has to be resident in main memory; therefore, we investigated how it scales for larger D and I by using synthetic data.

Figures 5.16 and 5.17 show how the access tree is affected by the vocabulary size, $|I|$ and the size of the database $|D|$. An interesting observation is that for smaller vocabularies, where the queries take longer to evaluate due to the existence of larger lists, the size of the access tree is smaller, too. This means that we can create HTI indices with larger thresholds to counter this effect. As the vocabulary increases, the *maximum* size of the trie augments superlinearly, thus, for large vocabularies the access tree tends to increase in a proportional way to the database size. For small vocabularies, the size of the access tree grows sublinearly (or remains stable if the maximum size has been reached) with respect to the database size. This is evident in Figure 5.18, where we vary the size of the database while keeping the vocabulary cardinality at 5k and the HTI threshold at 0.5. In the respective experiment with $|I| = 1k$ the tree reaches its maximum size (31 nodes) very soon and remains invariant to the size of D.

5.5.3 Threshold choice

Whereas the vocabulary and the database size depend on the data we have, the threshold for the HTI index is a choice we must make according to the speed requirements and the memory we have at our disposal. To highlight its effect we created several HTI indices for different thresholds and we show their performance in Figure 5.19 by varying the threshold from 0.2% to 10%. We depict simultaneously how the access tree grows, in 1000's of nodes, and how the average disk page accesses for the three types of queries fall as the threshold grows. After a certain threshold the average disk pages accesses are not significantly reduced, whereas the size of the access tree continues to grow, even if not as fast as for very low threshold.

5.6 Summary

In this chapter we presented the HTI index, a novel index that combines a trie with an inverted file to provide superior performance for containment queries. The key idea of the HTI is to break up the larger inverted lists to many small ones, that contain known combinations of the most frequent items. The combinations are kept in a main memory trie that leads to the disk resident inverted lists. A significant advantage of this approach is that it is complementary to the compression of the inverted lists, which is the basic enhancement that is used to ameliorate the performance of the inverted files. Our experiments showed that the HTI significantly outperforms, often by orders of magnitude, the inverted files, for all the types of

queries we examined. The results show that the HTI addresses efficiently the performance problems of the inverted files when dealing with skewed distributions or with very large databases with a small vocabulary.

The cost we pay for the HTI comes from two factors: a) the main memory we need to store the access tree and b) the increased update cost. As we have shown, a small amount of main memory is enough to provide significant performance gains to the HTI, thus the main memory requirement is not prohibitive for the HTI. Moreover, the update costs are only increased by a constant factor with respect to the inverted file. This factor is affected basically from the required additional pass over the data. Overall, both of these factors have a limited negative effect, thus the increased performance of HTI can be exploited in a variety of real world applications.

In our view the basic limitation of the HTI is the empirical tuning. In our experiments we had to refine the percentage of the items that are kept in the access tree, by testing. Whereas this might not be a significant problem in most application areas, the ideal solution would be able to automatically break up the longest of the inverted lists to small inverted lists, whose size falls in the desired bounds. Motivated by this, we propose in the next chapter the *ordered inverted file*, a new index that does not require tuning and offers superior performance in query evaluation, with respect to the HTI, for any distribution, at the cost of an even more increased creation and maintenance time.

Chapter 6

The Ordered Inverted File index

6.1 Introduction

In Chapter 4 we have shown how the inverted file can be used to answer containment queries on large collections of low cardinality set-valued data, and why its performance suffers, when the domain is limited or when the distribution of items is skewed. In Chapter 5 we have proposed the HTI index to address the problem of skewed distributions. The HTI breaks the larger inverted lists to smaller ones, that contain known combinations of the most frequent items of the database. This way, the HTI handles effectively collections of data that are dominated by few frequent ones. Still, its tuning relies on some expert who will decide how many items should be indexed by the access tree in order to have the maximum gain for the available memory. Moreover, the performance gain over the simple inverted file diminishes as the item's distribution moves closer to uniform.

The aforementioned results show that there is room for further improvement on the evaluation performance of containment queries. To this end, we propose a novel indexing scheme, the Ordered Inverted File OIF. The OIF indexes the inverted lists, but, differently from previous proposals, orders the records and exploits this ordering to efficiently evaluate containment queries. The crux of the approach is that, due to this ordering, there is a theoretical possibility of identifying a subset of the search space that needs to be checked. The OIF does not index every record *id* in the inverted lists, but instead it indexes blocks of record *id*'s, according to the *value* of the last record. Such an indexing scheme, allows to transform containment queries to range queries over the ordered records. As a natural solution, OIF employs B-trees to index the inverted lists. In this chapter we make the following contributions:

- We propose a novel indexing scheme, the ordered inverted file OIF, which outperforms the current state-of-the-art for containment queries on low cardi-

nality data from a limited domain. Our proposal is simple to implement and provides superior performance in all cases.

- We propose effective methods of creating and maintaining the OIF.
- We provide new evaluation algorithms for subset, set-equality and superset queries that take advantage of the proposed index.
- We theoretically investigate the performance of the OIF for the aforementioned queries and we show that it scales significantly better than inverted files. Moreover, we assess our proposal by extensive experiments on both real and synthetic data.

6.2 The Ordered Inverted File

In this section we describe the structure of the ordered inverted file (OIF). The OIF index has two basic parts: (a) an inverted file, created after suitable ordering of the records and, (b) a B-tree on top of the inverted file. As we saw in Section 4.2.1, indexing the inverted lists does not help in query evaluation time significantly, except if the number of candidates is very small and the lists that must be examined substantially large. However, if the candidate *id*'s are numerous and their distribution at the remaining inverted lists is unknown, it is usually preferable to sequentially access the whole lists, than to use the index to check the existence of each candidate. The approach, is similar to join by merge-sort in traditional databases. In the case of the OIF, this drawback is leveraged; by ordering the record *id*'s in the inverted lists, and by suitably tagging blocks of *id*'s in secondary storage, we limit the search space in a specific part of the inverted lists. Moreover, we create a sparse B-tree keeping only *one* key for each memory page of the inverted lists. This way, the space overhead is small, without significantly compromising query performance.

6.2.1 Index Structure

The ordered inverted file is based on the introduction of an ordering for the database items and records. Examples of possible orderings include the ordering by item frequency, lexicographic value etc. Later in the this chapter, we demonstrate that when containment queries are posed, this ordering allows the identification of the specific area in the inverted lists that contains the answer to subset, equality and superset queries. By coupling this property with a B-tree that provides direct access to all disk pages of each inverted list, we are able to significantly decrease disk page accesses in query evaluation.

In terms of structure, the *ordered inverted file* (OIF) index comprises the following:

1. An inverted file, where the inverted lists contain references to the database records, according to a special ordering.

2. A collection of B-trees, one for each inverted list, which contain pointers to each disk page of the respective list. The B-tree is built by considering the last record referenced in each disk page as a key.

Ordering of the inverted lists. As already mentioned, the suitable ordering of the records within the inverted lists of the inverted file can improve the processing of containment queries. The ordering of the records is based on the ordering of the items of I. Let the support $s()$ be a function that returns how many times an item appears in database D. Then, for any two items $o_i, o_j \in D$:

$$o_i >_D o_j = \begin{cases} true & if \quad s(o_i) > s(o_j) \\ true & if \quad s(o_i) = s(o_j) \wedge o_i >_a o_j \\ false & \text{otherwise} \end{cases} \tag{6.1}$$

where $>_a$ stands for alphabetic order. Equation 6.1 provides the definition of a total order operator $>_D$ based on frequency. The only practical difference from ordering the items of I according to their support is that by using $>_D$ we have a total order for I and not a partial one.

Based on the order for I we define the *sequence form*, sf, of a set value.

Definition 6.2.1. *Given a set value v from a database D, with n items we define as the sequence form of v, the sequence $sf(v) = o_1, \ldots, o_n$ where $o_i >_D o_j$ for $i < j$.*

Now we can define a partial order between the sequence forms of the set values. Let $sf(v_i) = o_i, o_{i+1}, \ldots, o_{i+n}$ and $sf(v_j) = o_j, o_{j+1}, \ldots, o_{j+m}$ be the sequence forms of two set values from database D, then:

$$sf(v_i) > sf(v_j) = \begin{cases} true & if \quad \begin{array}{l} o_{i+k} > o_{j+k} \text{ and} \\ o_{i+l} = o_{j+l},\ l < k \end{array} \\ false & \text{otherwise} \end{cases} \tag{6.2}$$

Note that we consider *null*, i.e., no item, as greater than all other items, similarly to the assumption of the common alphabetic order. The operators $<, =$ are defined with the obvious semantics based on Equation 6.2.

Using this ordering we can assign *id*'s to records in descending order, i.e., the record with the highest sf, will have the first *id* (equalities can be solved randomly

tid	products sold
1	$\{a, b, c\}$
2	$\{a, b, c, d, e, g\}$
3	$\{a, b, c, d, t\}$
4	$\{a, b, d\}$
5	$\{a, b, e\}$
6	$\{a, b, e\}$
7	$\{a, b, f\}$
8	$\{a, c, d\}$

tid	products sold
9	$\{a, c, f\}$
10	$\{a, c, g\}$
11	$\{a, e\}$
12	$\{a, f\}$
13	$\{b, c, d\}$
14	$\{b, d\}$
15	$\{c, g\}$
16	$\{e\}$

Figure 6.1: *A collection of set values in an object-relational database*

by using the order or appearance/processsing). Fig. 6.1 shows the database of Fig. 4.2 after the records have been ordered and assigned a new *id*. This way, for two records t_1, t_2, with $t_1.id < t_2.id$, holds that $sf(t_1.s) \geq sf(t_2.s)$. Depending on the requirements of the application we can create these *id*'s in two ways: (a) by using intermediate arrays that associate logical *id*'s with physical links or (b) by placing the record themselves sequentially in the hard disk following a descending order; in this case we can directly treat the physical links as *id*'s. In the former case the *id*'s of Fig. 6.1 are enough to directly access the records, in the latter the *id*'s of Fig. 6.1 must be "translated" to physical addresses in order for a record to be retrieved. This is often the way simple inverted files work [FBY92].

The query evaluation algorithms we propose in Section 6.3 can be used with other orderings, e.g., pure lexicographical or numeric. Still, we choose to order each inverted list based on the frequency of appearance (support) of each item in the database D for two reasons: (a) records containing frequent items that are expected to appear in queries' answers more frequently than the rest, are placed in the beginning of the inverted lists and (b) the longest lists contain record *id*'s that have small distances between them; this will lead to better compression, since compression techniques are based on the distance of sequential *id*'s in inverted lists.

Tagging for inverted lists. Knowing that records are placed in a descending order in the inverted lists does not guarantee that we can quickly identify the part of the inverted list where the answer to a query lies. Assuming that qs is an arbitrary set value provided by the user, we must be able to compare it with the records we encounter as we scan each inverted list. To this end, we employ a *tag* mechanism that provides the lower bound for each memory page. Each disk page is associated with a page tag, which is the sf of the last record that is referenced in the disk page. This way, we can avoid fetching the next page from the hard disk, if we can infer that it is unnecessary.

Again we can store the tags in two ways: (a) store them along with the inverted lists in the secondary storage and (b) store them in another index (we use a B-tree),

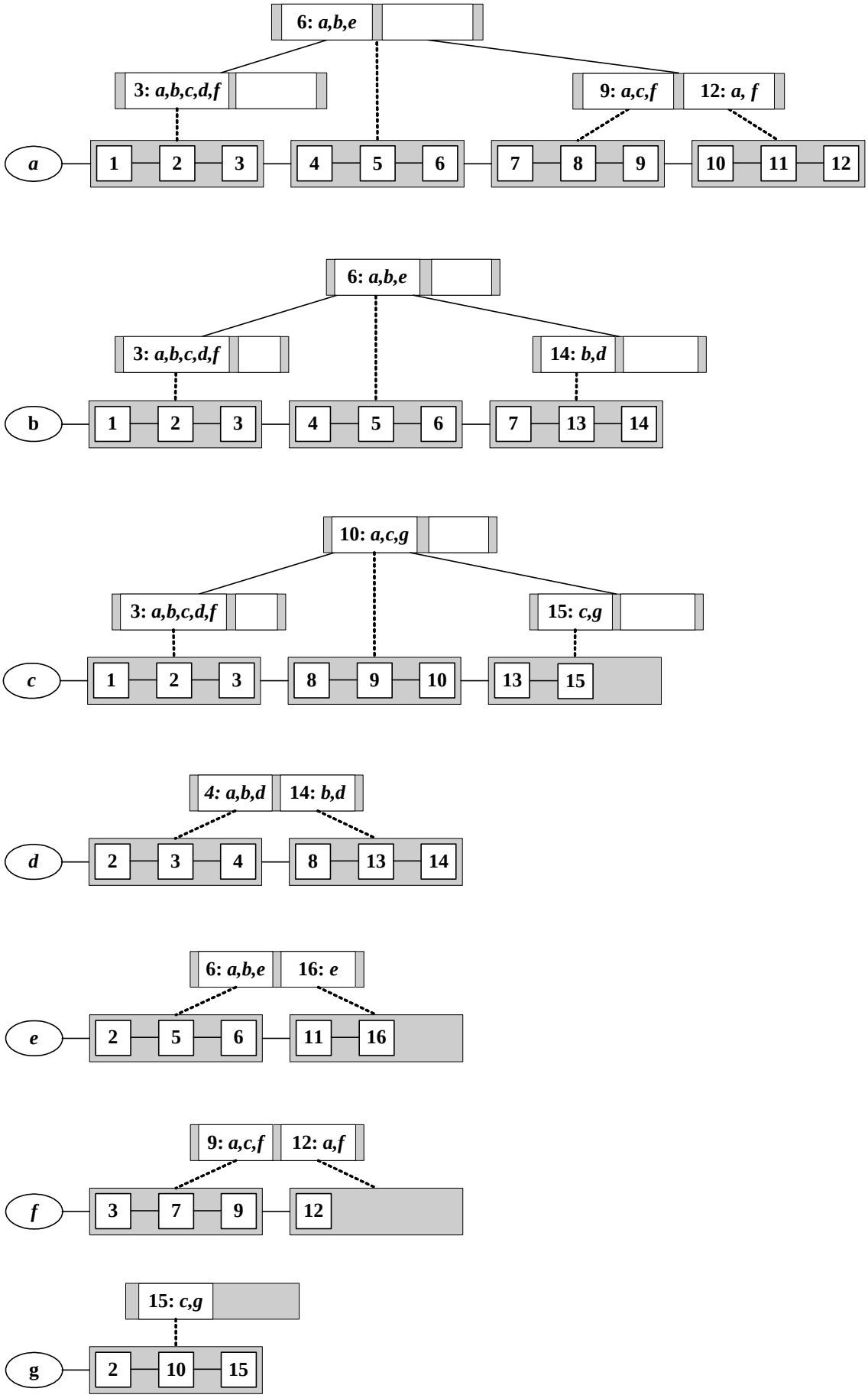


Figure 6.2: *OIF for the database of Fig. 6.1.*

that can fit in main memory. As usually, there is a tuning decision to make here that concerns the equilibrium between the benefits from keeping the tags in an index as opposed to the memory overhead this index imposes.

B-trees for inverted lists. We use a B-tree to store the tags associated with the disk pages for the inverted list of each item of the vocabulary. Each entry in the B-tree has three parts: (a) the tag, which is the *key* of the B-tree entry, (b) the *id* of the record, which we use heuristically in the subset query and (c) the physical address of the associated disk page (depicted as a dotted edge in Fig. 6.2). An OIF index for the database of Fig. 6.1 is depicted in Fig. 6.2.

Each B-tree has as many entries as the number of disk pages of the associated inverted list. The size of each entry depends on the size of the record. Thus, if the average record length is n, and each of the (i) $t.id$, (ii) item value and (iii) physical address has a size k, then, each entry has the size of $n\times$sizeof(item id)+sizeof($t.id$)+sizeof(pointer) $= (n + 2) \times k$. Moreover, if we compress the inverted list to $x\%$ of its initial size, we reduce the entries of the B-tree to $x\%$ of the original.

6.3 Query evaluation

In simple inverted files, where the *id*s of the records have no special meaning, the answer to an arbitrary query is usually scattered throughout the inverted lists of the involved items. The evaluation of set containment queries over inverted files, resolves to the computation of merge joins between the inverted lists [HM03]. Thus, in order to retrieve the entire answer we usually have to scan the entire lists. The case of superset is the worst, since the inverted lists have to be scanned more than once. Even if we provided links to intermediate points in the inverted lists via some index, the effect would be limited; there would be some gain in using the index, only when we would be interested in locating a single or very few *id*'s in the list. Unfortunately, this is not the case for merge-joins between long lists.

On the other hand, having the records ordered in the inverted lists as we proposed in Section 6.2, allows the identification of a region of the search space that contains the answer to each query. We call such a region *Range of Interest* (RoI) of the query. The knowledge of the theoretic bounds of a query's RoI enables the efficient usage of the B-tree index; instead of trying to locate numerous *id*'s that are candidate members of the answer, we only retrieve links to the first and last page of the part of the inverted list that contains such candidates.

The algorithms for the evaluation of the subset, set equality and superset queries against a database indexed by correlated inverted files have two basic steps:

1. The identification of the RoI.

2. The suitable merge-join of the inverted list regions that cover the RoI.

The difference from the evaluation of the same queries against simple inverted files lies at the first step. The algorithms for processing the retrieved record id's are virtually the same; it is the pruning of the inverted lists that brings the major improvement.

The bounds of RoI can be computed based only on the qs and the type of the query predicate. Having identified the theoretical bounds of RoI, we know the broadest possible region of the search space that contains the answer. To benefit from this knowledge we must be able to efficiently identify the corresponding part of the actual data, i.e., the disk pages of the inverted list that contain the RoI. This is feasible by utilizing the tags that are associated with the disk pages. We trace the sequence of disk pages whose tags cover the RoI by using the B-tree that has been constructed over these tags. The first page of this sequence is the first page in the list that has a tag greater or equal to the lower bound of the RoI and the tag of the last one must be strictly greater than the greater bound of RoI. We denote this region as $\lfloor RoI \rfloor$.

In other words, whereas the RoI stands for the maximum theoretical range of set values that must be searched each time, $\lfloor RoI \rfloor$ stands for the region of disk pages that must be retrieved from the hard disk for the algorithm to execute correctly.

In the rest of this section, we present the evaluation algorithms for each query predicate and we define the RoI for each case (each time denoted as RoI_{sub}, RoI_{eq} and RoI_{sup}). Moreover, we show that examining the RoI's is adequate to trace the entire answer to each query.

6.3.1 Subset

We define RoI_{sub}, i.e., the RoI for subset queries as follows:

Definition 6.3.1. Range of Interest for subset queries ($\mathbf{RoI_{sub}}$)*Given a subset query with a query set* $qs = \{o_{1^q}, \ldots, o_{n^q}\}$ *from a domain* I=$\{o_1, \ldots, o_N\}$, $qs \subseteq I$ *with* $o_{1^q} < o_{i^q} < \cdots < o_{n^q}$, *the Range of Interest* RoI_{sub} *is a region with the lower bound being the sequence* $o_1, o_2, \ldots, o_{n^q}$ *and the upper bound the sequence* $o_{1^q}, o_{2^q}, \ldots, o_{n^q}, o_N$.

1^q stands for the rank in I of the first item of the query, 2^q for second and so on.

Theorem 6.3.1. *It is sufficient to search for records within* RoI_{sub} *to answer a subset query.*

Proof. It is easy to see why the theorem holds by contradiction. Assume that there is one record $t = \{o_1', \ldots, o_m'\}$ that appears before the RoI_{sub}. Then it must hold,

that some $o_i^{'} < o_i$ or $o_i^{'}$ does not exist and o_i exists, with o_i being the i-th item of the lower bound of RoI_{sub}. Since the i-th item of the lower bound is the i-th item of I, $o_i^{'} < o_i$ does not hold, thus only if $\{o_1^{'}, \ldots, o_m^{'}\}$ is a prefix of the lower bound of RoI_{sub} can t appear before RoI_{sub}. But in this case it will not contain the last item of the lower bound, which is the last query item. Thus t will not satisfy the query. Similarly, it can be proven that t cannot appear after the greater bound of RoI_{sub} □

Assume for example that we would like to retrieve all the records that contain $qs = a, b$ from the relation of Figure 4.2. Then, $RoI_{sub} = [(a, b), (a, b, z)]$ (assuming that z is the smallest item of I). It is easy to see that in the order of Equation 6.2, the first candidate record has an $sf = a, b$ and the last $sf = a, b, z$, where z the last item in I. Any record t with $sf(t) > sf(\{a, b\})$ does not contain b and any record t' with $sf(t') < sf(\{a, b, z\})$ lacks a or b or both.

At the same time, the $\lfloor RoI \rfloor$ in the inverted list of a will be the first three disk pages, since the last one has a tag greater than the greater bound of RoI_{sub}. Similarly, the $\lfloor RoI \rfloor$ of item b will be again the first three disk pages.

The algorithm for evaluating the subset query by exploiting OIF is depicted in Figure 6.3. Similarly to the case of the inverted file it is a simple merge-join. Still, the behavior of OIF in both the average and the worst case is significantly better, since in the case of OIF we only intersect the $\lfloor RoI \rfloor$ part of the inverted lists, whereas in the case of inverted files we have to intersect the entire inverted lists.

Subset Query Evaluation
// $qs = \{o_1, \ldots, o_n\}$, $o_1 < \cdots < o_n$.
1. Determine the RoI for qs
2. for each i in $\lfloor RoI \rfloor$ of $o_n.list$ {
3. for ($j = n - 1; j \geq 1; j - -$) {
4. found=false;
5. while ($c < i$) {
6. get next item c from o_j.list;
7. if ($c = i$) then found=true }
8. if (not $found$) break }
9. if ($found$) output i; }

Figure 6.3: *OIF Subset Evaluation Algorithm*

The algorithm takes advantage of the fact that under the assumptions we made about the statistical properties of the data, where $|I| \ll N$, the RoI_{sub} is always positioned very close to the beginning of the involved inverted lists. In the previous example, the lower bound of RoI_{sub} was $sf(a, b)$, which is the second greatest sf that can appear in D (with a being the first). Therefore, without compromising

the correctness of the computed results, we can assume that the RoI_{sub} is always a *prefix* of the inverted lists, for all practical purposes.

Subset performance analysis. To evaluate the performance of subset queries, in the OIF index, we need to be able to estimate the size of RoI_{sub}. Assume a $qs = o_{q_1}, \ldots, o_{q_n}$ with $o_{q_i} >_D o_{q_j}, i < j$ and that each item o_i has a probability p_i of appearing in D. Moreover, assume that there is no special correlation between any of the items. First we have to estimate the number of records whose sf starts with prefix o_{i_1} in the list of item o_m, $i_1 < m$:

$$S_{<o_{i_1}>} = p_{i_1} \times \left(1 - \sum_{j=1}^{i_1-1} p_j\right) \times (p_m \times |D|)$$

In the same list, the number of records that start with o_{i_1}, o_{i_2},

$$S_{<i_1,i_2>} = S_{<i_1>} \times \left(p_{i_2} \times \left(1 - \sum_{j=i_1+1}^{i_2-1} p_j\right)\right)$$

and for $o_{i_1}, \ldots, o_{i_n}$,

$$S_{<i_1,\ldots,i_n>} = S_{<i_1,\ldots,i_{n-1}>} \times \left(p_{i_n} \times \left(1 - \sum_{j=i_{n-1}+1}^{i_n-1} p_j\right)\right)$$

or

$$S_{<i_1,\ldots,i_n>} = (p_m \times |D|) \times \prod_{l=1,\ldots,n} \left(p_{i_l} \times \left(1 - \sum_{j=i_{l-1}+1}^{i_l-1} p_j\right)\right)$$

Now, if we want to find what is the offset OS for the $qs = o_{q_1}, \ldots, o_{q_n}$ with $o_{q_i} >_D o_{q_j}$, we must add the S of all paths of length 1 that precede $< o_{q_1} >$, all the paths of length 2 that precede $< o_{q_1}, o_{q_2} >$ etc:

$$\begin{aligned}
OS_{<o_{q_1},\ldots,o_{q_n}>} &= \sum_{i=1}^{q_1-1} S_{<o_1>} + \cdots + \sum_{i=q_{n-1}+1}^{q_n-1} S_{<o_{q_1},\ldots,o_{q_n-1},o_i>} \\
OS_{<o_{q_1},\ldots,o_{q_n}>} &= (p_m \times |D|) \times \left[\sum_{i=1}^{q_1-1} p_i \times \left(1 - \sum_{j=1}^{i-1} p_j\right) + \cdots \right. \\
&\quad \left. + \sum_{i=q_{n-1}+1}^{q_n-1} \left(p_i \times \left((1 - \sum_{j=q_{n-1}+1}^{q_i-1} p_j\right)\right) \times \prod_{l=1,\ldots,n-1} \left(p_{q_l} \times \left(1 - \sum_{j=q_{l-1}+1}^{q_l-1} p_j\right)\right)\right] \\
&= (p_m \times |D|) \times \left(\sum_{k=1}^{n} \left[\sum_{i=q_{k-1}+1}^{q_k-1} \left(p_i \times \left((1 - \sum_{j=q_{k-1}+1}^{q_i-1} p_j\right)\right)\right.\right. \\
&\quad \left.\left. \times \prod_{l=1,\ldots,k-1} \left(p_{q_l} \times \left(1 - \sum_{j=q_{l-1}+1}^{q_l-1} p_j\right)\right)\right]\right) \qquad (6.3)
\end{aligned}$$

assuming $q_0 = 1$.

The most important term of the previous formula is for $k = 1$, which gives the offset of all the paths that start with the o_{q_1}, and it can be effectively used as an approximation. Using only this term the offset of RoI_{sub} is 0 and its end is at $OS_{<o_{q_1}>}$, which, assuming a uniform distribution, is equal to:

$$\begin{aligned} OS_{<o_{q_1}>} &= \sum_{i=1}^{q_1-1} p_i \times \left(1 - \sum_{j=1}^{q_1-1} p_j\right) \times (p_m \times |D|) \\ &= \sum_{i=1}^{q_1-1} p \times (1 - (i-1) \times p) \times (p \times |D|) \\ &= (p^2 \times |D|) \times \left[(q_1 - 1) - \sum_{i=1}^{q_1-1} (i-1) \times p\right] \\ &= (p^2 \times |D|) \times (q_1 - 1) \times \left[(1 - \frac{(q_1 - 2)}{2} \times p\right] \end{aligned}$$

Note, that this is not the least upper bound we used in the definition of RoI_{sub} but it gives a good estimation. In the worst case, where the greatest item of qs is at the last positions in I, the RoI_{sub} covers almost the whole inverted list, thus the performance of OIF is similar to the performance of the inverted file. On the other hand, as our experiments confirm, in the average case the RoI_{sub} significantly limits the region of the inverted list that is examined.

The fact that the RoI_{sub} is a prefix of the inverted list has an interesting consequence: there is no need to directly access any disk page but the first. We can sequentially access all the disk pages that overlap with RoI_{sub} without compromising the performance of the algorithm. As a result, if we are only interested in evaluating subset queries, there is no need to introduce any main memory overhead with reference to the inverted file; the page tags can be stored at secondary storage and we can retrieve them along with each disk page.

Finally, we use the B-tree heuristically: if the merge-join of the first lists that are considered has discarded many candidate solutions, we use the B-tree to search in the next inverted list, if we estimate that we can avoid merge-joining some parts of it. In our case, we use an estimation based on the density of the *id*'s in each list.

6.3.2 Equality

The range of interest RoI_{eq}, for the set equality query is a single point in the search space. We define it as follows:

Definition 6.3.2. Range of Interest for equality queries ($\mathbf{RoI_{eq}}$) *Given an equality query with a query set qs, the range of interest RoI_{eq} is a continuous region with lower and upper bound the $sf(qs)$.*

Intuitively, to evaluate an equality query, we trace the pages in each inverted list that contain the answer via the B-trees (plus an extra one in the case where a tag is equal to the $sf(qs)$) and we merge-join them. The evaluation algorithm is virtually the same with the subset evaluation algorithm depicted in Figure 6.3. The basic difference is that now we use the RoI_{eq} and that we filter records according to their length; only records that have length equal to $|qs|$ are considered in the merge join.

Equality performance analysis. When the data are indexed by a simple inverted file, the worst case (which is also the most common one) for the equality queries is the same as for the subset query; the entire inverted lists have to be examined. In the average case, query processing is slightly faster than in subset queries due to the usage of the length of the record as an additional filter. Also, in the case of equality queries, RoI_{eq} is very small compared to RoI_{sub}; therefore, the OIF index significantly accelerates the processing of equality queries. Actually, the false positives (i.e., record id's that are retrieved from the hard disk but are discarded in the evaluation procedure) that appear in $\lfloor RoI_{eq} \rfloor$ cannot be more than 2×(page size) in each inverted list. [1] As a result, the evaluation of the equality queries with OIF depends linearly on the size of the answer and on the size of the query $|qs|$ and only logarithmically on the size of the inverted list and consequently on the size of D. The logarithmic dependence on the size of the inverted list is a consequence of the traversal of the B-tree which takes place in the process of deciding the $\lfloor RoI \rfloor$ of the query. Since the height (thus, the maximum page accesses in a traversal) of the B-tree constructed over an inverted list of size n is $\log_{m/2}(\frac{n}{PS})$ (with m being the order of the B-tree and PS the page size), the page accesses in the worst case are:

$$\text{page accesses} = \sum_{i=1}^{|qs|} \log_{m/2}(\frac{n_i}{PS}) + |qs| \times \frac{\text{answer size}}{PS}$$

The order m depends on the size of the B-tree key, i.e. the average size of the record. If the selectivity is small and the B-tree is cached, then we only have $|qs|$ disk page accesses, i.e., one from each inverted list. This also agrees with the results of the experimental evaluation of Section 6.5. For the case of the simple inverted file, the disk pages accesses would be $\frac{(n_1+\cdots+n_{|qs|})}{PS}$.

[1] The worst case appears only when we have $(PAGE\_SIZE * i) + 1,\ i = 1, 2, \ldots$ answers in an inverted list and the first answer is the last record in a disk page (thus it is the tag of the page).

6.3.3 Superset

Superset queries return records that include only items that belong to the query set (i.e., the record is a subset of the query set). During the evaluation of superset queries, we progressively produce the answers to the query that contain each of the items of the qs. We examine the items starting from the most frequent one and continue examining them in descending order of frequency. In each step, all the answers that contain the current item are produced, except if they have already been identified in a previous step. Unlike the evaluation procedure for the subset and equality queries the Range of Interest for the superset queries is differentiated by the item and the iterative step.

Definition 6.3.3. *Given a query set $qs = \{o_1, \ldots, o_n\}$, where $o_1, \ldots, o_n$ are listed in descending order of frequency the RoI_{sup} of the first step is*

$$\begin{aligned} RoI^1_{sup-o_1} &= [S(o_1), S(o_1, o_n)] \\ &\vdots \\ RoI^1_{sup-o_i} &= [S(o_1, o_2, \ldots, o_i), S(o_1, o_i, o_n)] \\ &\vdots \\ RoI^1_{sup-o_n} &= [S(o_1, o_2, \ldots, o_n), S(o_1, o_n)] \end{aligned}$$

in the i-th step

$$\begin{aligned} RoI^i_{sup-o_i} &= [S(o_i), S(o_i, o_n)] \\ &\vdots \\ RoI^i_{sup-o_n} &= [S(o_i, o_{i+1}, \ldots, o_n), S(o_i, o_n)] \end{aligned}$$

and in the last step

$$RoI^n_{sup-o_n} = [S(o_n), S(o_n)]$$

In Figure 6.5, the RoI_{sup} for the case of $qs = \{a, c, f\}$ is depicted. To make it more easily interpretable we represent the records not with their id's but with all their items and depict all the bounds of the regions like existing records. As the Figure 6.5 implies, the following holds:

Theorem 6.3.2. *For a query set $qs = \{o_1, \ldots, o_n\}$ it is enough to examine all the RoI of Definition 6.3.3 for the identification of the answer to the query. qs.*

Proof. We can prove Theorem 6.3.2 by contradiction, following the steps of the proof for the Theorem 6.3.1. By taking a record $t = \{o'_1, \ldots, o'_m\}$, and assuming

that t does not fall into any of the areas defined in Definition 6.3.3, we can prove that t cannot be a valid answer to the superset query, because it must contain more objects that the objects of the qs. To see how this holds, one has to observe first that the merge-join of all RoI^i_{sup}'s, as defined in Definition 6.3.3 in the i-th step, is enough for identifying all the answers than contain o_i and have not been identified in a previous step. □

Superset Query Evaluation
// $qs = \{o_1, \ldots, o_n\}$, $o_1 < \cdots < o_n$.
1. for (k=1;$k \leq n$;k++) {
2. Determine the RoI^k's for qs
3. for each i in $RoI^k_{sup-o_k}$ of $o_k.list$ {
4. $tolMisses = n - (k-1) - o_k.length$
5. for ($j = k+1$;$j \leq n$;j++) {
6. if $tolMisses < 0$ break
7. $tolMisses - -$
8. for each item $c < i$ in $RoI^k_{sup-o_j}$ {
9. if ($c = i$) $tolMisses + +$}}
12. if $tolMisses \leq 0$ output i}}

Figure 6.4: *Inverted file Superset Evaluation Algorithm*

The algorithm for evaluating superset queries with the OIF index is depicted in Figure 6.4. The intersection between the inverted lists is based again on a merge-join as in the case of the subset and equality evaluation algorithms. What differentiates the superset algorithm is that it can afford to miss an item if the length of the record is smaller than the qs. The idea is that the record might be comprised of any items of the qs. Thus if we have a record of length 3 and a qs of length 10, we can afford to miss 7 items before deciding that the record is not a valid answer.

Superset performance analysis. The basic steps of the algorithm for evaluating superset queries over a simple inverted file resemble the algorithm of Fig. 6.4 for evaluating superset queries with OIF. The only difference of the two algorithms is that the algorithm for inverted files considers items of the full inverted list, instead of considering only items from the RoI_{sup}. However, as we show in the following, this improvement affects the complexity of the algorithm significantly. In the case of the inverted file, in each iteration of the outer loop we have to scan the whole inverted lists of the remaining items. Since one item is removed from the merge join at each iteration, the worst case requires $n_1 + 2n_2 + \cdots + |qs|n_{|qs|}$ disk pages accesses, where n_i is the size of inverted list of i-th item, in disk pages. In the following, we establish that the worst case scenario for the OIF is $n_1 + n_2 + \cdots + n_{|qs|}$ disk page accesses.

Lemma 6.3.1. *For an item o of qs the RoI^i_{sup-o} in step i, starts after the end of RoI^{i-1}_{sup-o} of step $i-1$.*

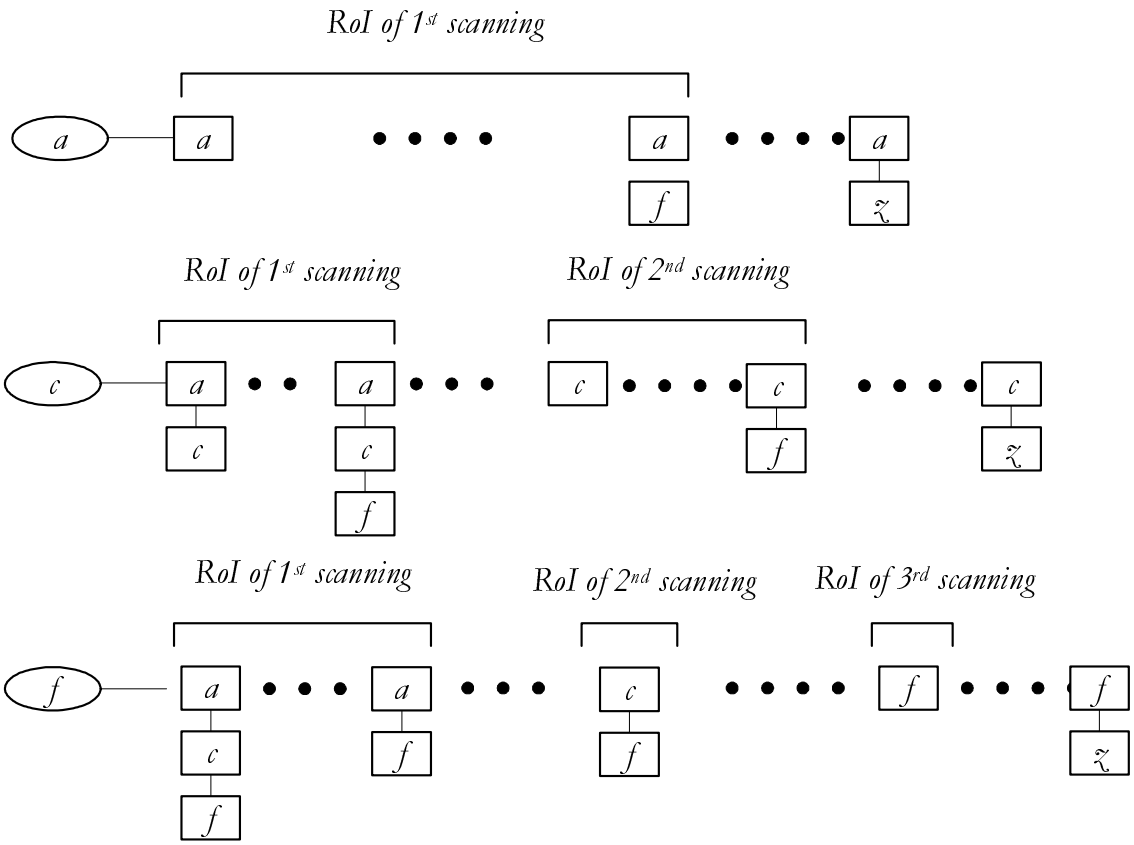


Figure 6.5: *The ranges of interest for the superset query* $\{a, c, f\}$.

It is easy to see why the lemma holds by observing the bounds of the *Roi*'s in Definition 6.3.3. Based on the previous lemma we can establish the following theorem:

Theorem 6.3.3. *The evaluation of a superset query requires at most scanning the inverted lists of all items once.*

Proof. The theorem comes as an immediate consequence of Theorem 6.3.2 and Lemma 6.3.1; if scanning the areas of RoI_{sup} is enough to find all the answers, and if at each step the algorithms scans non-overlapping subareas of RoI_{sup}, at the end of the evaluation RoI_{sup} is scanned only once. □

Note that Theorem 6.3.3 defines the worst case. By using the B-tree to directly access the several different *RoI*'s we have an average case that is often significantly better than the worst. This superior theoretical behavior of the superset evaluation algorithm in OIF is fully reflected by our experimental results as we show in Section 6.5.

6.4 Creation and Maintenance

6.4.1 Creation

To create an OIF index, one more processing stage is required compared with creating a simple inverted file: sorting the records. Sorting the records by using as a key all their values is a tedious task but it can be addressed quite effectively by a *radix sort* based algorithm. When dealing with large collections of data, the implementation of radix sort is not straightforward, since main memory will probably not be sufficient to store the whole dataset. If we apply a radix sort algorithm, without any provision for memory management, we will end up with many read-writes in the secondary storage, which will dominate the processing time. A straightforward way of implementing a radix sort for the kind of data we have is to split the data into buckets by first considering their most frequent item each time. This way, we will know that all the set-values in the first bucket should get smaller *id*'s that those of the second and so on. The problem of such partitioning is that it does not take into consideration the size of the resulting buckets. It follows that the first buckets will be the biggest, even for a uniform distribution, and they will probably require retrieving and rewriting them to the secondary storage several times, as we further split them into more buckets. To achieve the most efficient sorting it would be best to be able to partition the data, in a single pass, to fixed sized buckets that can fit into main memory. To do so, we propose a variation of a trie-based radix-sort [SZ04, SZ03].

Creating a trie by using all the data, would sort everything in a single pass, but it is unlikely that the trie will fit into main memory, and it will introduce a significant secondary storage I/O cost. Our basic idea is to create a trie where each path will lead to buckets of similar size, that will fit into main memory. To achieve this, we employ the analysis we performed in Section 6.3.1 for the complexity of subset queries. The equation 6.3 does not only provide the offset for a specific sequence in an inverted list, but it can be used to estimate where a specific sequence will be positioned in an ordered database. Using this equation, we can determine prefixes that act as boundaries for (estimated) fixed size buckets. Each bucket hold records that are greater that the prefix of the previous bucket but smaller that the prefix of the current bucket. All prefixes are kept in a main memory trie and at the end of each path a link is given to each bucket. If the main memory is not large enough to hold the whole trie, then we can create larger buckets with smaller prefix-boundaries, and then split each bucket again following the same algorithm. The basic steps of such algorithm are the following:

1. Estimate the probability of appearance of each item, either exactly by scanning

once the whole database, or by sampling.

2. Determine the number of buckets that will be created and calculate the maximum prefix that can be hold in a main memory trie.

3. Estimate the boundaries for each bucket.

4. Scan the database and place each record in the bucket with the matching prefix boundary.

5. If the resulting bucket fits into main memory, retrieve and sort it in main memory, else go again to step 2 and split it to smaller buckets.

Estimating the boundaries for each bucket can be done by using the Equation 6.3. If we adjust Equation 6.3 to give the offset of each sequence in the whole database the result would be:

$$OS_{<o_{q_1},\ldots,o_{q_n}>} = |D| \times \left(\sum_{k=1}^{n} \left[\sum_{i=q_{k-1}+1}^{q_k-1} \left(p_i \times \left((1 - \sum_{j=q_{k-1}+1}^{q_i-1} p_j \right) \right) \times \prod_{l=1,\ldots,k-1} \left(p_{q_l} \times \left(1 - \sum_{j=q_{l-1}+1}^{q_l-1} p_j \right) \right) \right] \right. \tag{6.4}$$

To estimate the boundaries between the buckets we need to find sequences that are estimated to appear (or just to be ranked) in the dataset in every b records, where b the size of the bucket in records. To find these sequences we first compute the probability of appearance for each item in a record and we estimate the offset of the records that start with this item in an ordered database. We create buckets of size $b \pm m$ by grouping together items that appear sequentially. We require that the offset for the records whose sf start with the last item of the bucket is estimated to appear after $b \pm m$ records from the first record, whose sf starts with the first item of the bucket. If adding an item makes the size of the bucket to got directly from a value less than $b - m$ to a value greater than $b + m$, we refine our grouping criterion by considering the second item that appears in the sf of a record. If the same problem appears again we continue with the third and so on, until we get the desired bucket size. [2]. As a boundary for each bucket we use the last sequence that belongs to its group. The algorithm is shown in Figure 6.6.

6.4.2 Updates

Handling updates in the OIF requires answering efficiently two additional challenges, with respect to updates in a simple inverted file: (a) how to insert a record

[2]This might not always be possible if some items appear only in certain combinations. In practice we use some additional safety constraints to avoid going into very large depths

Determine bucket boundaries
// Let b be the size of the bucket and m the approximation factor.
// Let S the current probability of items, $coff$ its offset
// and $loff$ the offset of the previous saved value of S.
1. determine the probability of appearance p_i of each item of the database
2. set $loff = 0$ and $coff = 0$.
3. while S not empty {
4. set $S = \emptyset$;
5. for each item o of I {
6. add the item o to the end of the current sequence S
7. $coff = coff + f_i$
8. if $(coff - loff < b - m)$ {
9. remove the last item of S
10. continue }
11. if $(b - m \leq coff \leq b + m)$ {
12. store S in the trie
13. $loff = coff$
14. remove the last item of S
15. continue }
16. if $(coff - loff > b + m)$ {
17. replace the last item of S with the previous item of o
18. continue }
19. }
20.}

Figure 6.6: *The algorithm for determining bucket boundaries takes the frequencies of the databases items as an input and outputs a trie that has the boundaries for buckets of size* $b \pm m$

in the right place in an the inverted list and (b) how to give to new records *id*'s analogous to their rankings. These two points are not as difficult to handle as a first approach might suggest. When updating an inverted file, we usually do batch updates as we explained in Section 4.5. Individual postings are accumulated in a small main memory inverted file, which is merged with the disk based memory file when there is no more memory available or when it is more convenient for each application. This way the first problem is rather easily tackled; if the lists are already indexed by a B-tree and they are already retrieved in main memory for the merging, the difference between adding the new *id*'s at the end of the list, or in intermediate places is not significant. Either by merge sort or by using the B-tree, depending on how many values must be inserted, such functionality can be achieved efficiently.

Updating the ranking of records and providing new *id*'s is more complicated and poses three basic problems: (a) efficiently sorting the updated database, (b) assigning *id*'s to all records whose rank changed, and (c) reflecting all the changes of the records *id*'s in the inverted lists. Efficiently answering the first problem is facilitated by keeping the initial trie in secondary storage. By retrieving and using the trie we can quickly decide which of the original buckets must be updated with the new data. Having identified the affected buckets and the data that must be inserted in them, we sort only locally the affected buckets and avoid re-sorting the whole

database. Giving *id*'s that reflect this new ranking can be simplified by originally assigning *id* that are not sequential, e.g. instead of 1,2,3, we can assign to the first records the *id*'s 10,20,30. This scheme can postpone the problem of sliding the *id* of existing records if the distribution of the items in the updates is similar to the original distribution. A more effective solution is to introduce *landmarks*, i.e. special marks every k records, and the assign *id*'s relatively to these marks. For example instead of 1,2,3,4,5,6 we can have l_1 1,2,3, l_2 1, 2, 3. A side effect of this solution, is that we must store the landmarks in the inverted lists too, in order to have a universal translation of each *id*. This might introduce significant overhead in the smaller lists. Finally, updating the inverted lists is simpler due to the compression techniques. Since we represent it's *id* with respect to the *d*-gap from the previous *id* in the list, if we have a whole block of *id* that has been "shifted" due to insertions, we only have to increase the first *d*-gap of the block. For example let 2,4,6,7,11,16,17,20 be one inverted list that is coded as 2,2,2,1,4,5,1,3. Assume that one more record has been inserted and, despite the aforementioned techniques, it replaced the record with $id = 3$, and the record with $id = 3$ got as new $id = 4$, and the record with $id = 5$, got $id = 6$ and so on. The inverted list must now become 2,5,7,8,12,17,18,21, this means that all but the first entries must change. But when we code it using *d*-gaps we only have to make 1 change: 2,3,2,1,4,5,1,3.

In any case, maintenance in the OIF is harder than maintenance in the simple inverted files. Still, since updates happen in a batch manner, by retrieving the whole index from the disk to merge it with the temporal in-memory index, the difference in the updating cost is limited.

6.5 Experimental Evaluation for the OIF index

To evaluate the performance of our proposal we compare it to the state-of-the-art solution for containment queries, the inverted file.

6.5.1 Methodology

Implementation. We implemented both the simple inverted file and the OIF in C++, in (a) an initial disk-based and (b) a memory-based version. Initial results proved, as expected, that the containment problem is dominated by disk page accesses (the evaluation of sets of queries took seconds when everything was in main memory and minutes when everything was on the disk), thus we performed the experiments with memory-based version for reasons of convenience. We kept the inverted lists in series of arrays equal to the page size we assumed (4k) and we explicitly counted each visit. The vocabulary was stored in a hash table in main

memory.

We used the same implementation for the simple inverted file and the inverted file part of the OIF, thus our results are directly comparable. Since compression techniques benefit the OIF at least as much as the simple inverted file, we did not use any compression since the behavior of our results would not change.

Data. We evaluated the OIF using again the two real datasets from UCI KDD repository [HB99], we introduced in 5.5. To map the behavior of the OIF as several statistical properties of the data vary, we also used synthetic data containing transactions with length varying from 2 to 24. We considered datasets sizes of 100k, 250k and 1M transactions from a domain of 2k, 5k and 10k objects. We created three datasets for each combination of database size and domain by following a different zipfian distribution each time. The zipfian parameters we used were 0, 0.5 and 1, thus ranging from uniform to skewed. The results we present in the following are always aggregated on all sets, except if stated otherwise.

Queries. As in other approaches [HM03], we evaluated our proposal using queries that always have an answer, considering them more informative than those that do not. We created such queries by using existing transactions, selected uniformly from all D. The selectivities of the subset queries are less than 3%, with the less selective being those with $|qs| = 2$. The most common case for larger $|qs|$ and for equality queries is that there are less than 5 answers. On the other hand the selectivity of superset queries can surpass 3% for large $|qs|$ on the real data. To provide accurate results we created 50 queries of each size and type.

6.5.2 Performance evaluation

In figures 6.7-6.11 we depict the evaluation results for the OIF and the simple inverted file. The results for the inverted file are labeled as *inverted*. For the case of the OIF we depict the disk page accesses for three different settings: (a) The index is clustered and the B-tree is on main memory. This is the case when all disk page accesses come from the inverted lists. We label this setting as *oif-inv*. (b) The index is clustered but the B-tree is also disk resident. We label this setting as *no-cache*. (c) The index is not clustered, thus we need an intermediate table to translate the *tid*'s to disk addresses and everything (the inverted file, the B-tree and the intermediate table) are disk resident. This is the worst case for the OIF and we mark it as *no-cluster*.

Subset. The OIF outperforms the simple inverted file both for real and for synthetic data. Moreover, it exhibits two very good properties. As the length of the query set grows and thus, queries have less answers, it manages to use the B-tree efficiently and detect fast the areas of interest in each list. This way its performance

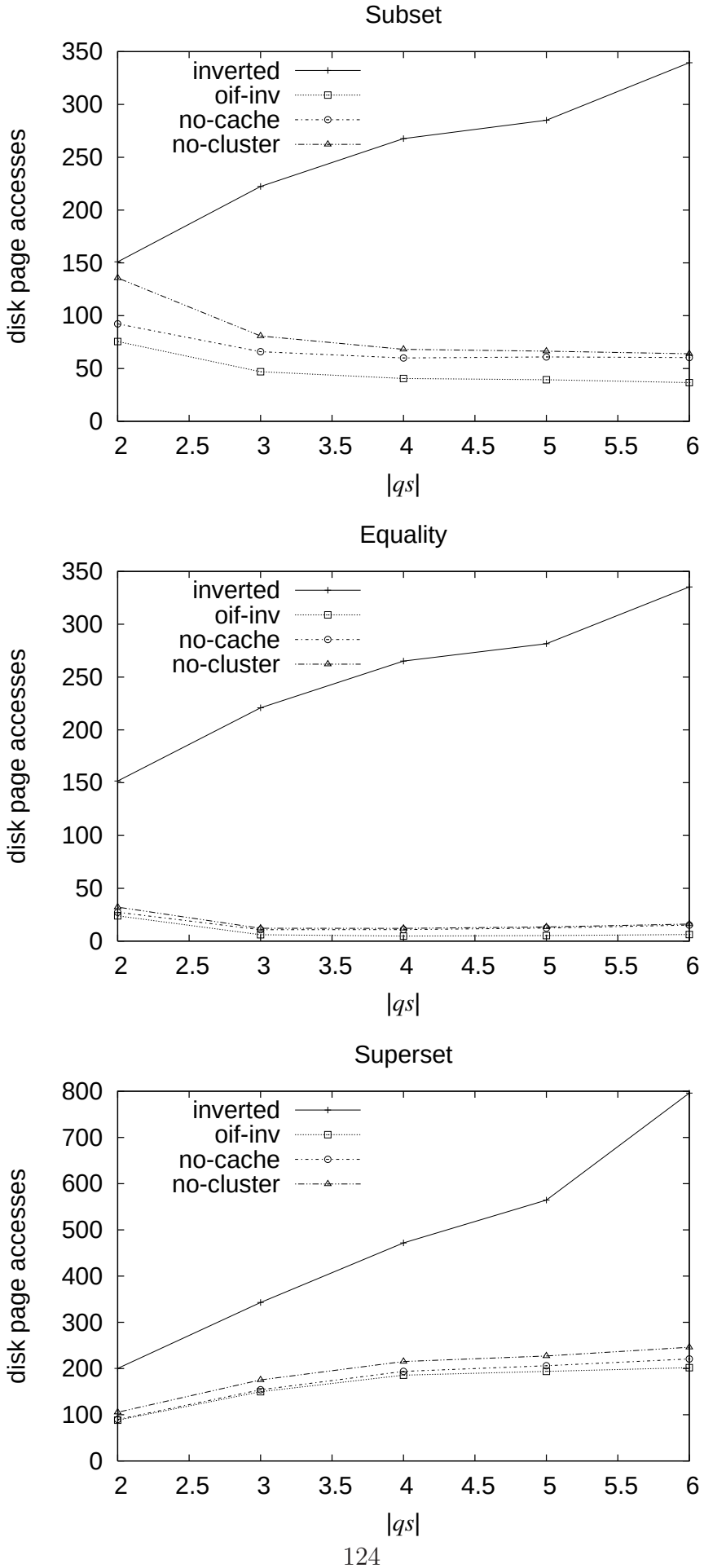


Figure 6.7: *Average performance of queries on msweb data*

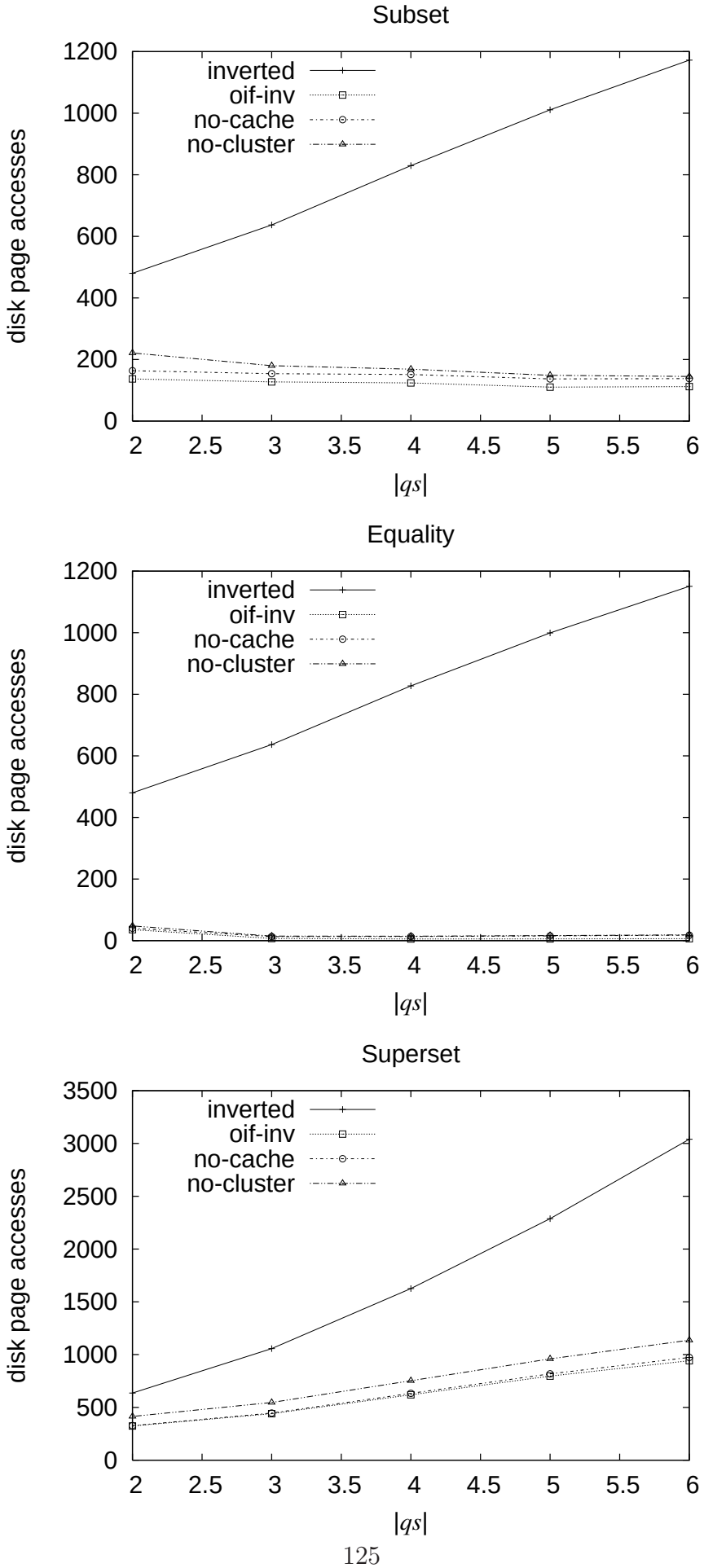


Figure 6.8: *Average performance of queries on msnbc data*

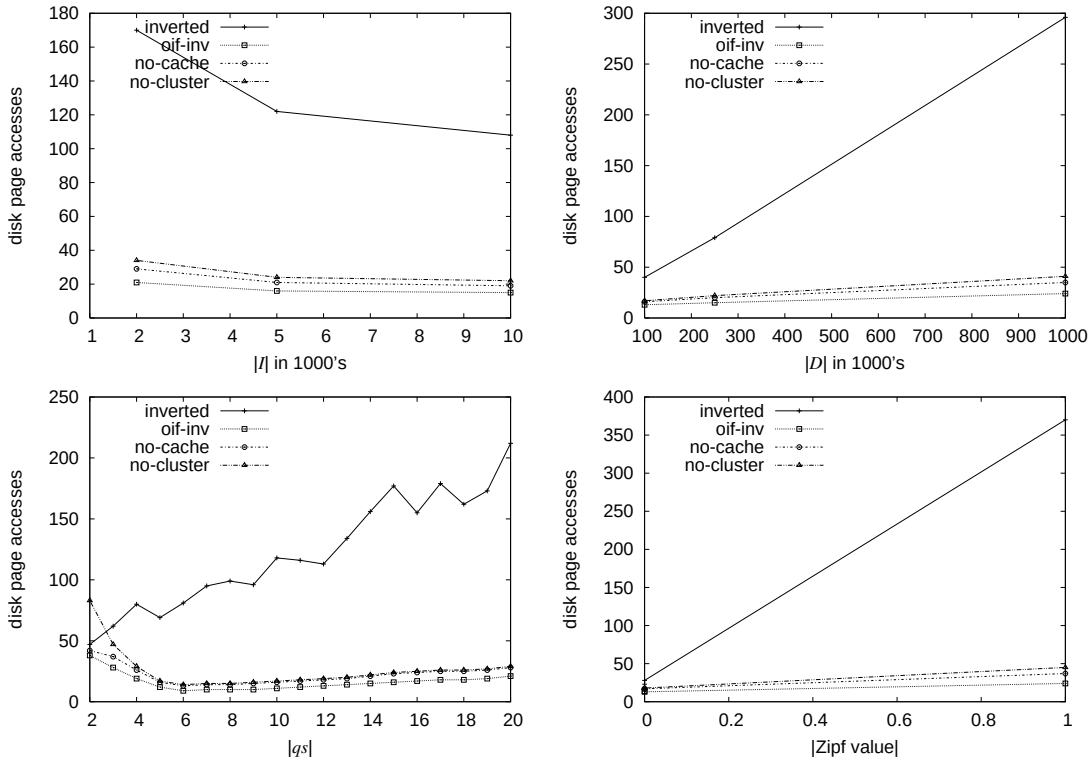


Figure 6.9: *Average performance of subset queries*

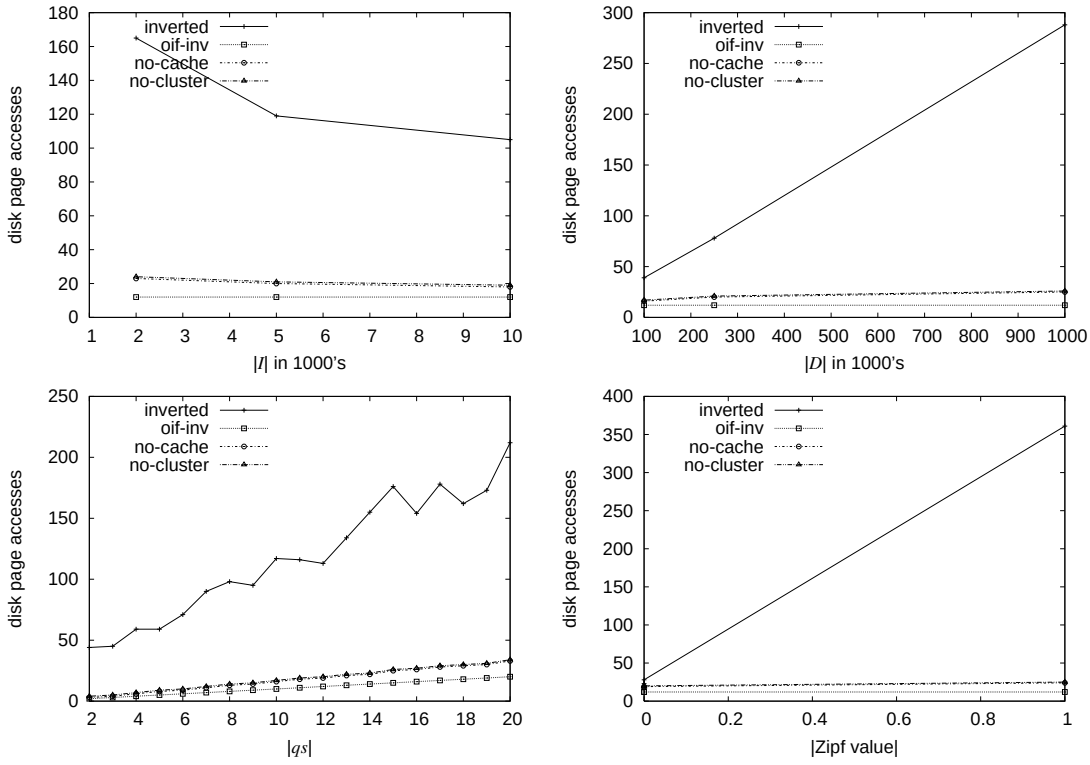


Figure 6.10: *Average performance of equality queries*

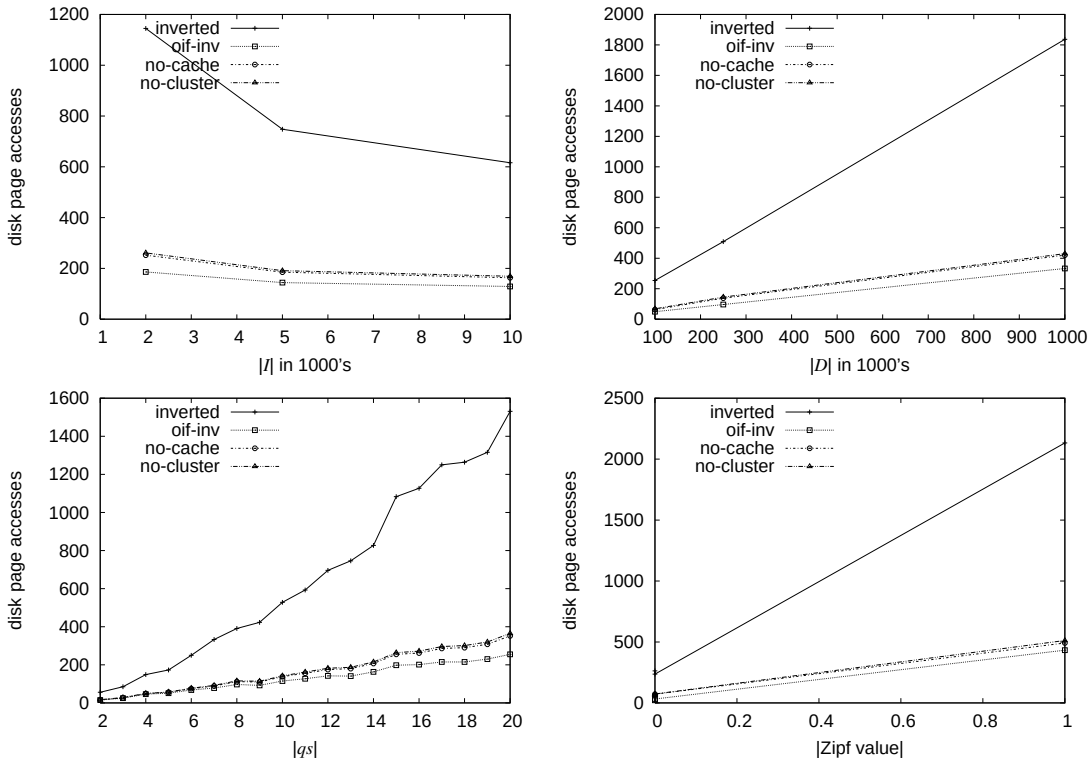


Figure 6.11: *Average performance of superset queries*

grows, unlike the case of the simple inverted file, which suffers when it has to examine many inverted lists and cannot benefit from the small selectivity. The other good property of the OIF is its scaling capabilities as the database D grows. As shown in Fig. 6.9, OIF scales a lot better than the inverted file. This is again attributed to the B-tree and the limited RoI.

Equality. As expected from the complexity analysis, the performance of the OIF is practically constant since it can accesses directly the disk pages where the possible answer lies. Experiments on both real and synthetic data confirm this.

Superset. Superset is by far the most expensive query studied, thus the performance gain in this case is more significant. As we can see from Figures 6.7, 6.8 and 6.11 the OIF greatly outperforms the inverted file in both real and synthetic data. Moreover, is scales a lot better both for large D and qs.

In summary, the OIF significantly outperforms the inverted file in all cases, for all different implementations settings (i.e. clustered or not clustered, with only exception being the subset query of length 2, where the size of the results forces many disk page accesses in the intermediate table). Moreover, it is unaffected by the skewness of the data, unlike the inverted file whose performance deteriorates as the skewness grows. Finally, the memory overhead we are paying for this is not large; for the case of the longest inverted lists ($|D| = 1M,\ |I| = 2k$) the B-tree has 10% of the size of the inverted file. This result comes from using a key of constant size, equal to the greatest transaction. By using a key of variable length, we should

get a B-tree with about half of size of the original.

6.6 HTI vs. OIF index

Having experimentally compared both the OIF and the HTI index with the inverted file, naturally arises the question about how they compare one against the other. A comparison between the OIF and the HTI has to take into consideration three factors: a) the memory overheads, b) the construction and update time and c) query performance.

6.6.1 Memory Overheads

Both the HTI and the OIF introduce some memory overheads. In the case of the OIF it is the sparse B-trees that are created for each list and in the case of the HTI it is the main memory access tree. It should be noted that the latter can be stored in secondary storage in various ways, for example it can be kept in a B-tree with each path being a separate key value. In both cases the overheads are small and the do not do constitute a decisive factor for choosing one over the other. Moreover, the space overheads of its methods can be tuned; by altering the number of items that are kept in the access tree, for the case of the HTI, and by choosing a different size of block for the OIF.

6.6.2 Construction and update time

The creation and update procedure that we proposed for the HTI differs from the respective one of the inverted file in two ways: a) it has to handle a bigger number of smaller lists and to create the access tree in main memory and b) it has to do an additional scan of the database due to indexing combinations of the most frequent items. Both of these operations are not very costly and preliminary results show that by even when using naive strategies the cost of building the HTI is around double of building the inverted file.

The case is not the same for the OIF. Using a radix sort to sort all records, might be an $O(|D| \times |\bar{t}|)$ procedure, but in practice it is quite expensive, since data have to be read and re-written in disk during the sorting stage.

Construction and update time constitute a clear advantage of the HTI over the OIF index. The faster update times, make the HTI more suitable for applications that handle data, which change frequently or in applications, where updates must be performed very quickly.

6.6.3 Query performance

Comparing the performance of the the HTI and the OIF in query processing is not trivial. The HTI, as expected, has a different behavior depending on the distribution of items in D. We compare the two indices with respect the number of disk accesses they perform during evaluation, against two datasets that follow a uniform distribution and a skewed Zipfian distribution of order 1. This is done, in order to see how they behave in the two ends of the spectrum. We create both the uniform and the skewed dataset with with 1M records of 2-22 items, with average size of 12 items from a vocabulary of 2K items. We create again subset, equality and superset queries following the methodology we presented in Section 6.5. We evaluate the OIF against three versions of the HTI, which index in the access tree the 0.5%, 1% and 3% of the top frequent items. Moreover we depict in Figures 6.12 and 6.13 the respective results for the inverted file as a point of reference.

The overall conclusion from the experimental results is that the OIF is the clear performance winner. Still, the results differ a lot depending on the items' distribution. In the case of uniformly distributed items, depicted in Figure 6.12 the HTI does not give any real advantage over the inverted file, and the OIF is the undisputable winner. When the items of the database follow a skewed distribution, the picture is drastically different. The OIF surpasses all versions of the HTI for equality and subset queries (Figure 6.13) and is second only the HTI with the largest access tree (3% of the top frequent items are kept in the access tree) for superset queries. Still, the performance gain over any of the versions of the HTI is not very big.

The previous analysis and experimental results show that the OIF offers a significantly better performance that the HTI at the cost of increased creation and maintenance costs. Moreover the OIF offers a very good performance independently of the items distribution. Still, when the items of the database follow a skewed distribution in the various records, the HTI offers a competitive performance at a significantly lower creation and maintenance cost.

6.7 Summary

In this chapter we presented the OIF index, which is a novel approach to indexing for containment queries. The basic idea of the OIF index is to introduce an order for all the items and the records of the database and to organize the inverted lists according to this order. Moreover, OIF employs a B-tree to offer intermediate access points in the inverted lists. As a result the search space of each query can be limited to a specific region in the inverted lists of items that participate, which can

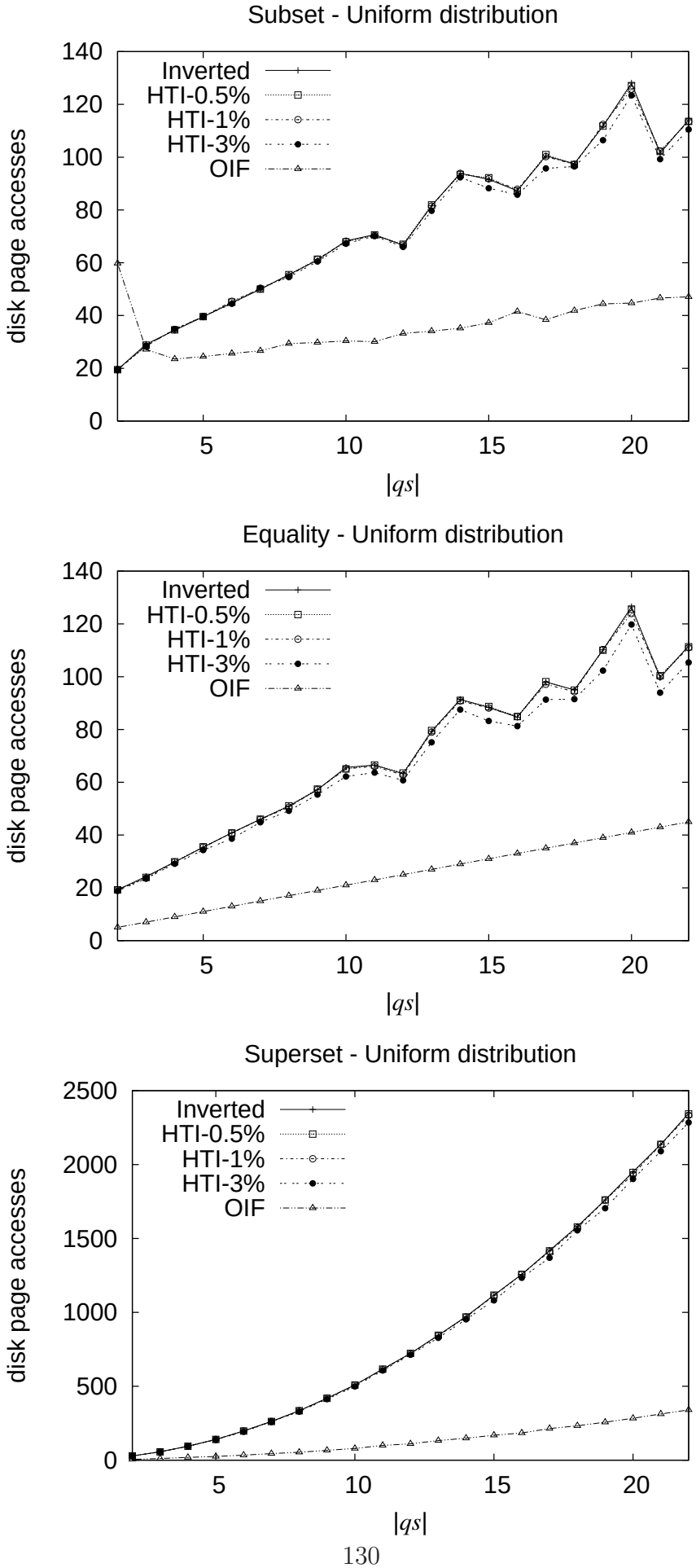
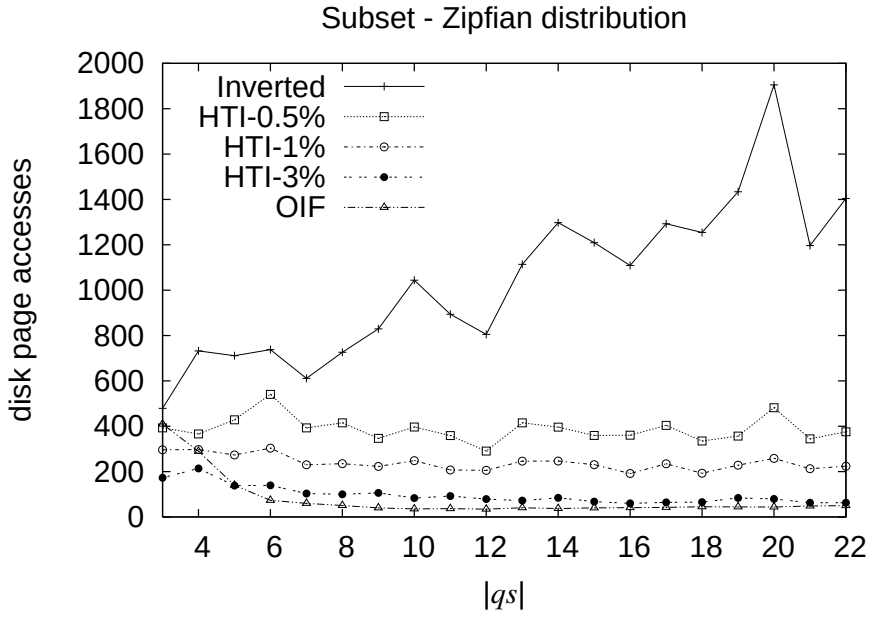
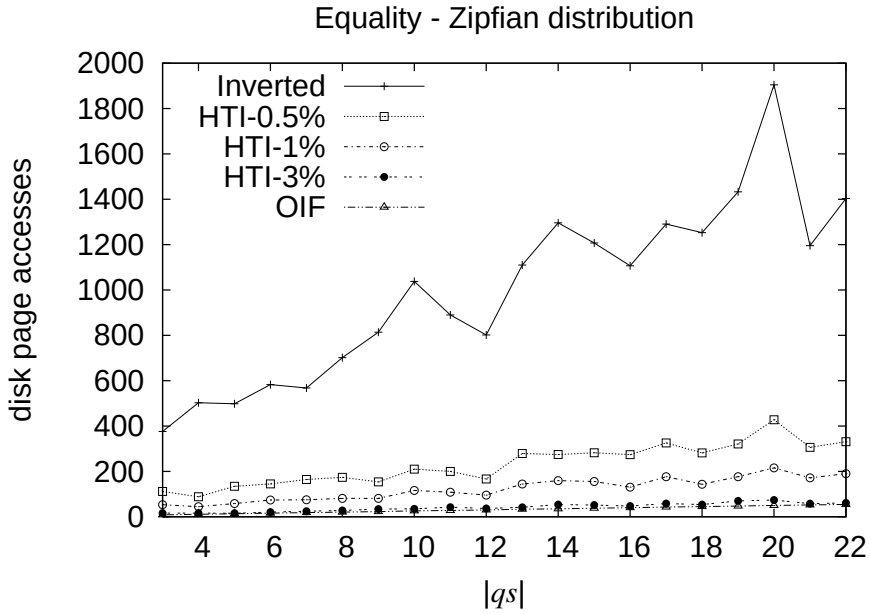


Figure 6.12: *Average performance of queries on uniformly distributed data*





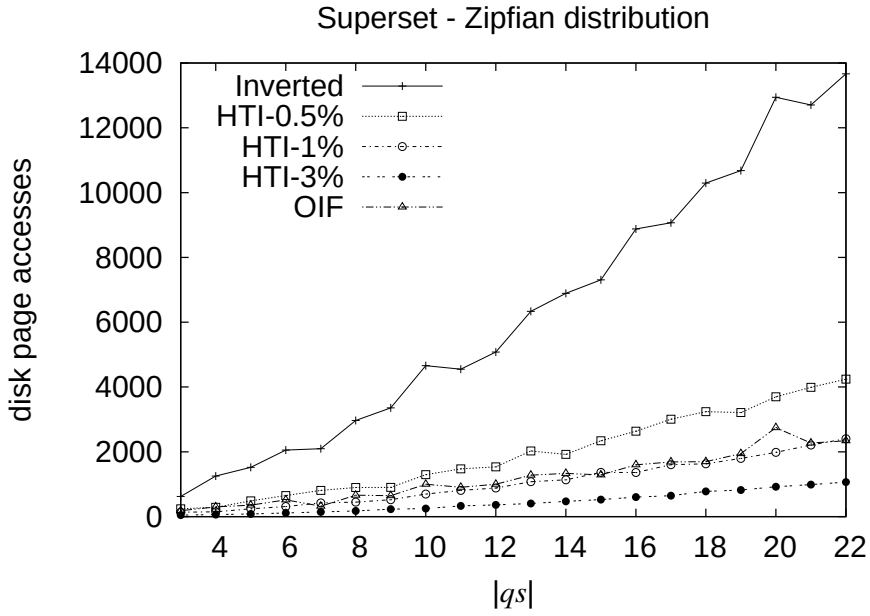


Figure 6.13: *Average performance of queries on data with a skewed item distribution*

be subsequently accessed via the B-tree of the inverted list.

The experimental results for the OIF showed that it provides a clearly superior performance over both the inverted file and the HTI index, often in orders of magnitude. Moreover, the OIF exhibits a more robust behavior in comparison with the HTI, since it provides a significant performance gain, independently of the items' distribution. This superior performance comes at the cost of an increased construction and maintenance cost. The core of the construction and maintenance costs lies in the need to sort all the records of the database. To counter this cost, we proposed a trie based radix sort, that has a complexity of $O(|D| \times |\bar{t}|)$ and significantly limited disk I/O.

The OIF index can be the suitable choice for systems that are not frequently updated, or can be updated offline but require a very efficient query evaluation mechanism.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis, we have identified and addressed the need to manage condensed and rich in semantics representations of raw data, termed patterns, in an way similar to how data are handled in modern RDBMSs. We have set the goal to support a unified environment which efficiently manages both data and patterns. In our solution we have approached the problem from two different ends. On one end, we have started from the high level requirements, and we have proposed a conceptual and logical model for handling patterns and data. On the other end, we have started from existing technology and have provided more efficient solutions for a very basic set of containment queries that are essential in building and supporting the pattern management environment. More specifically, we have set and addressed the following requirements:

1. *We have required that the PW would allow the handling of both data and patterns, and that it would preserve the complex semantics of patterns.* To address this goal we have proposed a conceptual architecture and a logical model that covered both data and patterns. The proposed model handles both data and patterns as primary citizens, but also as distinct entities. The semantics of the patterns are expressed by their relation to the underlying data.

2. *We wanted to have a transparent representation of pattern semantics, which would facilitate the creation of some reasoning mechanism and allow the users to combine them to create new patterns.* To this end, we have proposed using a logical language for expressing the semantics of patterns, i.e., the relation between patterns and data. The language that expresses this relation can be different, depending of the application domain. In the context of this thesis, we have adopted a constraint-based language, which is suitable for

expressing patterns that are conceived as regions in a metric space. Finally, we have exposed how patterns can be used to discover properties of the data by proposing a sound and complete model for binary pattern relations that can support reasoning mechanisms for data.

3. *We have required that the Pattern Warehouse would preserve the relation between the patterns and the underlying data.* To this end, we have proposed two ways of representing this relations; an approximate one, by describing the image of the patterns in the data space in a logical language, and explicit one, where patterns and data are explicitly linked by records. We have complemented this work by presenting operators, which demonstrate the gains of preserving the pattern-data relation, by facilitating the easy navigation from the pattern to data space and vice versa.

4. *We have required that the Pattern Warehouse would implement efficiently the navigation between patterns and data.* In this context, we have focused on the efficient representation of the explicit relation between data and patterns. Our contribution is not limited in the framework of PW, and applies generally to indexing techniques for containments queries. Our proposal consists of the Hybrid Trie-Inverted file index and the Ordered Inverted file index. Both these indices address the weaknesses of the inverted file which is the current state-of-the-art for containment queries. Such indices are required if we need to efficiently identify which data are represented by which patterns, and vice-versa which patterns represent a specific set of data. Both indices address the generalized problem of containment query evaluation, thus our results have also impact on existing database and information retrieval technology. The HTI tackles the problem of highly skewed distributions by sacrificing a small amount of main memory. A basic advantage of the method is that the performance gain is mainly affected by the size of the domain unlike traditional cashing techniques, which are affected by the size of the database. The OIF on the other hand, offers superior query performance for all queries independently of the items' distribution, but it has an increased creation and maintenance cost, since the underlying data must first be sorted. Both of these methods, significantly reduce the cost of evaluating containment queries in very large databases, thus making possible the creation of systems like the PW, which need to efficiently navigate between the pattern and the data space.

This thesis augments our understanding of the problem of handling patterns and provides new indices that efficiently address the evaluation of containment queries. Still, bridging the gap between current object-relational systems, and a system that

will be able to handle uniformly and generically the results of complex data analysis methods, poses numerous challenges. In the following section, we provide a sketch of future work, which is tightly linked to the research results of the thesis.

7.2 Future Work

As most research works, this thesis opens as many questions as the issues it addresses. The Pattern Warehouse setting, as presented in this thesis, opens new research challenges both in its modelling and in its implementation. The most important points, we have identified for future work, are the following:

- In the context of the logical model of the Pattern Warehouse we have proposed a series of operators, for querying patterns and data. Still, these operators do not constitute an algebra, since no guarantees about their expressivity and safety are provided. An imperative completion to this work, would be to provide a full algebra, based on these operators, which will provide the necessary guarantees.

- Until now, our research efforts were focused on clusters and in general patterns which correspond to contiguous regions in some metric space. Studying more use-cases is necessary, if we want to provide a robust solution that will be able to handle a variety of patterns. In this context, patterns like graphs in non-metric space, biomedical data etc., need to be investigated. Moreover, we need to examine if we can use a more generic language for the mapping formula, that will be able to accommodate such patterns, or an entirely new language is needed for such application areas.

- In our study of the Pattern Warehouse, we have assumed that patterns are created by some external mechanism or at least in a black-box manner. It would be an interesting challenge to see if we can incorporate some basic mining mechanisms, which will be able to work for a variety of different data types.

- The access methods for set values we have presented in the Chapters 5 and 6, provided a significant speed up in the evaluation of containment queries. Still, we did not investigate a very important family of query operators, which might benefit from our proposals: set-containment joins. A basic goal of our future work, is to investigate if we can use the ideas we have developed in the HTI and OIF indices for making the evaluation of containment joins more efficient.

- Containment queries are a very common form of queries emerging in a variety of applications. We aim at investigating how the basic techniques we developed in the context of the HTI and the OIF can be exploited to answer more effectively queries on graphs and biomedical data, where paths are sequences of data values are often the target of the queries.

- Finally, we aim to complement our work on containment query processing by considering the case of multi-query optimization, i.e., the evaluation of complex containment predicates. We also plan to investigate how these techniques can be adjusted to work on streaming data, where the constraints are significantly different from the case of static data.

In general we believe that there is a plethora of interesting research issues that are open in the implementation of the Pattern Warehouse and in the exploitation of the HTI and the OIF. Our future and ongoing efforts aim at pursuing a deeper understanding of the issues that are linked to pattern management and containment query evaluation.

Bibliography

[AB95] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.

[AdKM01] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, New York, NY, USA, 2001. ACM Press.

[AM98] Vo Ngoc Anh and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*, pages 290–297. ACM, 1998.

[AM02] Vo Ngoc Anh and Alistair Moffat. Impact transformation: effective and efficient web retrieval. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–10, New York, NY, USA, 2002. ACM Press.

[AWY99] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu. A new method for similarity indexing of market basket data. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 407–418. ACM Press, 1999.

[Bay71] Rudolf Bayer. Binary b-trees for virtual memory. In *Procceedings of the ACM-SIGFIDET Workshop, San Diego, California*, 1971.

[BCC94] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 192–202, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[BCN+04] I. Bartolini, P. Ciaccia, I. Ntoutsi, M. Patella, and Y. Theodoridis. A unified and flexible framework for comparing simple and complex patterns. In *Proceedings of 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'04), Pisa, Italy, September 20-24, 2004*, volume 3202 of *Lecture Notes in Computer Science*, pages 496–499. Springer, 2004.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331, 1990.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BMV96] Daniel Barbará, Sharad Mehrotra, and Padmavathi Vallabhaneni. The gold text indexing engine. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*. IEEE Computer Society, 1996.

[Bou04] J. F. Boulicaut. Inductive databases and multiple uses of frequent itemsets: the CINQ approach. In *Database Support for Data Mining Applications: Discovering Knowledge with Inductive Queries*, volume 2682 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2004.

[Bro95] Eric W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 30–38, New York, NY, USA, 1995. ACM Press.

[BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[CBM+94] T. R. Couvreur, R. N. Benzel, S. F. Miller, D. N. Zeitler, Dik Lun Lee, Mukesh Singhal, Niranjan G. Shivaratri, and Wai Yee Peter Wong. An analysis of performance and cost factors in searching large text databases using parallel search systems. *JASIS*, 45(7):443–464, 1994.

[CC95] C. Clarke and G. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system. Technical Report MT-95-01, Department of Computer Science, University of Waterloo, 1995.

[CCB94] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-94-40, University of Waterloo Computer Science Department, 1994.

[CCH92] James P. Callan, W. Bruce Croft, and Stephen M. Harding. The INQUERY retrieval system. In *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.

[CCKN01] Jin-Yi Cai, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. On the complexity of join predicates. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 207–214, New York, NY, USA, 2001. ACM Press.

[Che05] Yangjun Chen. On the signature trees and balanced signature trees. In *ICDE*, pages 742–753, 2005.

[CIN03] The CINQ project. http://www.cinq-project.org, 2003.

[CM05] J. Shane Culpepper and Alistair Moffat. Enhanced byte codes with restricted prefix properties. In *Proccedings of the 12th International Conference on String Processing and Information Retrieval, (SPIRE 2005)*, 2005.

[CP90] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411, New York, NY, USA, 1990. ACM Press.

[CWM01] Common Warehouse Metamodel (CWM). http://www.omg.org/cwm, 2001.

[Dep86] Uwe Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 77–87. ACM, 1986.

[DR02] L. De Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, 2002.

[Dun02] Margaret H. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice-Hall, 2002.

[EMHJ93] Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'93), May 25-28, 1993, Washington, DC*, pages 253–264. ACM Press, 1993.

[ESF01] M. G. Elfeky, A. A. Saad, and S. A. Fouad. ODMQL: Object Data Mining Query Language. In *Proceedings of International Symposium on Objects and Databases, Sophia Antipolis, France, June 13, 2000*, volume 1944 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2001.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Comput. Surv.*, 17(1):49–74, 1985.

[Fal92] Christos Faloutsos. Signature files. In *Information Retrieval: Data Structures & Algorithms*, pages 44–65. 1992.

[FBY92] William B. Frakes and Ricardo A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[FC84] Chris Faloutsos and Stavros Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.

[FC87] Christos Faloutsos and Stavros Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Trans. Inf. Syst.*, 5(3):237–257, 1987.

[FKM00] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 119–135, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[FL91] Edward A. Fox and Whay C. Lee. Fast-inv: A fast algorithm for building large inverted files. Technical report, Blacksburg, VA, USA, 1991.

[GGK01] Aristides Gionis, Dimitrios Gunopulos, and Nick Koudas. Efficient and tunable similar set retrieval. In *ACM SIGMOD*, 2001.

[GMUW00] Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *Database System Implementation.* Prentice-Hall, 2000.

[Gol66] S. W. Golomb. Run-length encodings. *IEEE Transaction on Information Theory*, 12(3), 1966.

[Gra02] Jim Gray. The information avalanche: Reducing information overload. http://research.microsoft.com/ Gray/Talks/, 2002.

[GRGL99] V. Ganti, R. Ramakrishnan, J. Gehrke, and W.-Y. Loh. A framework for measuring distances in data characteristics. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 126–137. ACM Press, 1999.

[GSBS05] Lin Guo, Jayavel Shanmugasundaram, Kevin Beyer, and Eugene Shekita. Efficient inverted lists and query algorithms for structured value ranking in update-intensive relational databases. In *Procceding of ICDE*, 2005.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[GW00] Roy Goldman and Jennifer Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *ACM SIGMOD*, 2000.

[HB99] S. Hettich and S. D. Bay. The uci kdd archive. university of california, department of information and computer science. 1999.

[Hel97] Sven Helmer. Index structures for databases containing data items with setvalued attributes. Technical Report 2/97, Universitat Mannheim, 1997.

[HFBYL92] Donna Harman, Edward Fox, Ricard Baeza-Yates, and Wahy Lee. Inverted files. In *Information Retrieval: Data Structures & Algorithms*, pages 44–65. 1992.

[HFW+96] J. Han, Y. Fu, W. Wang, K. Koperski, and O.Zaiane. Dmql: A data mining query language for relational databases. In *1996 SIGMOD'96 Workshop. on Research Issues on Data Mining and Knowledge Discovery (DMKD'96)*, Montreal, Canada, June 1996.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The International Journal on Very Large Data Bases*, 12(3):244 – 261, 2003.

[HP94] J.M. Hellerstein and A. Pfeffer. The *rd*-tree: an index structure for sets. Technical Report 1252, University of California, Berkley, 1994.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *Procs. VLDB*, 2002.

[HPYM00] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation. In *ACM SIGMOD*, 2000.

[HPYM04] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[IKO93] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobuo Ohbo. Evaluation of signature files as set access facilities in oodbs. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 247–256, New York, NY, USA, 1993. ACM Press.

[IM96] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[Ins06] Food Marketing Institute. Supermarket facts & figures. http://www.fmi.org/facts_figs/superfact.htm, 2006.

[IV99] T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, 2(4):373–408, 1999.

[JDM03] Java Data Mining API. http://www.jcp.org/jsr/detail/73.prt, 2003.

[JLN00] T. Johnson, L. V. S. Lakshmanan, and R. T. Ng. The 3W Model and Algebra for Unified Data Mining. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000), September 10-14, 2000, Cairo, Egypt*, pages 21–32. Morgan Kaufmann, 2000.

[JO95] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *ACM SIGMOD*, 2004.

[KKTR02] Manolis Koubarakis, Theodoros Koutris, Christos Tryfonopoulos, and Paraskevi Raftopoulou. Information Alert in Distributed Digital Libraries: The Models, Languages, and Architecture of DIAS. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, pages 527–542, Rome, Italy, September 2002.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KST06] Manolis Koubarakis, Spiros Skiadopoulos, and Christos Tryfonopoulos. Logic and computational complexity for boolean information retrieval. *IEEE Trans. on Knowledge and Data Engineering*, 18(12), 2006.

[LMZ05] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM Press.

[LV00] P. Lyman and H. R. Varian. How much information. http://www.sims.berkeley.edu/how-much-info, 2000.

[LWP+03] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic maintenance of web indexes using landmarks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 102–111, New York, NY, USA, 2003. ACM Press.

[LZW04] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[LZW06] Nicholas Lester, Justin Zobel, and Hugh E. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[Mam03] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In *ACM SIGMOD*, 2003.

[MB95] Alistair Moffat and Timothy A. H. Bell. In situ generation of compressed inverted files. *J. Am. Soc. Inf. Sci.*, 46(7):537–550, 1995.

[MCL03] Nikos Mamoulis, David W. Cheung, and Wang Lian. Similarity search in sets and categorical data using the signature tree. In *ICDE*, 2003.

[MGM03a] Sergey Melnik and Hector Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28(1):56–99, 2003.

[MGM03b] Sergey Melnik and Hector Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28(1):56–99, 2003.

[MMNM03] Mikolaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *Proceedings of 7th East European Conference on Advances in Databases and Information Systems*, Lecture Notes in Computer Science, pages 236–252. Springer, 2003.

[MPC99] R. Meo, G. Psaila, and S. Ceri. An Extension to SQL for Mining Association Rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1999.

[MRYGM01] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, 2001.

[MZ96] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[NM02] Alexandros Nanopoulos and Yannis Manolopoulos. Efficient similarity search for market basket data. *The VLDB Journal*, 11(2):138–152, 2002.

[NTCM02] Mario A. Nascimento, Eleni Tousidou, Vishal Chitkara, and Yannis Manolopoulos. Image indexing and retrieval using signature trees. *Data & Knowledge Engineering*, 43(1):57–77, 2002.

[PAN02] The PANDA Project. http://dke.cti.gr/panda/, 2002.

[PMM03] Predictive Model Markup Language (PMML). http://www.dmg.org/pmmlspecs_v2/pmml_v2_0.html, 2003.

[Ram01] Karthikeyan Ramasamy. *Efficient Storage and Query Processing of Set-Valued Attributes*. PhD thesis, Computer Sciences Department, Madison, Wisconsin, July 2001.

[RBC+03] S. Rizzi, E. Bertino, B. Catania, M. Golfarelli, M. Halkidi, M. Terrovitis, P. Vassiliadis, M. Vazirgiannis, and E. Vrachnos. Towards a logical model for patterns. In *Proceedings of 22nd International Conference on Conceptual Modeling (ER'03), Chicago, IL, USA, October 13-16, 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2003.

[RPNK00] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton, and Raghav Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 351–362, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[SC05] Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, 2005.

[SDKR+95] Ron Sacks-Davis, Alan J. Kent, Kotagiri Ramamohanarao, James A. Thom, and Justin Zobel. Atlas: A nested relational database system for text applications. *IEEE Trans. Knowl. Data Eng.*, 7(3):454–470, 1995.

[SH03] Justin Zobel Steffen Heinz. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8), 2003.

[SK04] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *ACM SIGMOD*, 2004.

[SM96] Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1996.

[SQL01] ISO SQL/MM Part 6. http://www.sql-99.org/SC32/WG4/Progression_Documents/FCD/fcd-datamining-2001-05.pdf, 2001.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518, 1987.

[ST96] A. Siblerschatz and A. Tuzhillin. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):970–974, 1996.

[STGM94] Kurt Shoens, Anthony Tomasic, and Héctor García-Molina. Synthetic workload performance analysis of incremental updates. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 329–338, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[Suc95] Dan Suciu. Domain-independent queries on databases with external functions. In Georg Gottlob and Moshe Y. Vardi, editors, *Proceedings of Database Theory (ICDT'95), Prague, Czech Republic, January 11-13, 1995*, volume 893 of *Lecture Notes in Computer Science*, pages 177–190. Springer, 1995.

[SWYZ02] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *ACM SIGIR*, August 2002.

[SZ03] Ranjan Sinha and Justin Zobel. Efficient trie-based sorting of large sets of strings. In *ACSC*, pages 11–18, 2003.

[SZ04] Ranjan Sinha and Justin Zobel. Using random sampling to build approximate tries for efficient string sorting. In *WEA*, pages 529–544, 2004.

[TBM02] Eleni Tousidou, Panayiotis Bozanis, and Yannis Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, 27(2):93–121, 2002.

[TGMS94] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 1994. ACM Press.

[The03] Yannis Theodoridis. The r-tree-portal, 2003.

[TIK05] Christos Tryfonopoulos, Stratos Idreos, and Manolis Koubarakis. Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks. In *Proceedings of the 28th Annual International ACM SIGIR Conference*, Salvador, Brazil, August 2005.

[TKD04] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. Filtering algorithms for information retrieval models with named attributes and proximity operators. In *ACM SIGIR*, 2004.

[TPVT] M. Terrovitis, S. Passas, P. Vassiliadis, and S. Timos. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM), Arlington, USA*.

[TVS+] Manolis Terrovitis, Panos Vassiliadis, Spiros Skiadopoulos, Elisa Bertino, Barbara Catania, Anna Maddalena, and Stefano Rizzi. Modeling and language support for the management of pattern-bases. *Data & Knowledge Engineering*. To appear.

[TVS+04a] M. Terrovitis, P. Vassialidis, S. Skiadopoulos, E. Bertino, B. Catania, and A. Maddalena. Modeling and language support for the management of pattern-bases. In *Proceedings of SSDMB 2004*, 2004.

[TVS+04b] Manolis Terrovitis, Panos Vassiliadis, Spiros Skiadopoulos, Elisa Bertino, Barbara Catania, and Anna Maddalena. Modeling and language support for the management of pattern-bases. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21-23 June 2004, Santorini Island, Greece*, pages 265–274. IEEE Computer Society, 2004.

[WL93] Wai Yee Peter Wong and Dik Lun Lee. Implementations of partial document ranking using inverted files. *Inf. Process. Manage.*, 29(5):647–669, 1993.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, 1999.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

[ZMSD92] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full text databases. In *VLDB*, 1992.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.