Efficient Answering of Set Containment Queries for Skewed Item Distributions

TR-IMIS-2010-1

Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis,
Timos Sellis, Nikos Mamoulis

Institute for the Management of Information Systems,
"Athena" RC, Greece

September, 2010

Abstract

In this paper we address the problem of efficiently evaluating containment (i.e., subset, equality, and superset) queries over set-valued data. We propose a novel indexing scheme, the Ordered Inverted File (OIF) which, differently from the state-of-the-art, indexes set-valued attributes in an ordered fashion. We introduce query processing algorithms that practically treat containment queries as range queries over the ordered postings lists of OIF and exploit this ordering to quickly prune unnecessary page accesses. OIF is simple to implement and our experiments on both real and synthetic data show that it greatly outperforms the current state-of-the-art methods for all three classes of containment queries.

Contents

Chapter 1

Introduction

Containment queries are meaningful whenever we need to examine membership properties (e.g. which records contain items a and b?) in collections of data. When posing a containment query we treat the underlying data as collections of sets, but data can be modelled in various ways; they can be set-values, they can span in several tuples of a relational table, or they can be XML documents with additional structural information. The efficient evaluation of containment queries is an important issue in several application areas, e.g., in market basket analysis the transactional logs of customers are examined to retrieve those that contain certain items. In this work we focus on three fundamental containment operators: subset, set-equality and superset and we propose an inverted file based index, which efficiently addresses skewed item distributions.

In the research literature there are two main classes of access methods specialized for supporting containment queries: signature files [14] and inverted file indices [25, 49]. Surveys have shown that inverted files outperform signature-based methods for containment queries on low cardinality set-values [21] and on text documents [48]. Moreover, inverted files have been shown to outperform traditional relational methods (B-trees) for containment queries in most cases [46]. Considering inverted files as the state-of-the-art mechanism for set containment is also supported by the fact that they are being used by all WWW search engines [45].

Nevertheless, the performance of inverted files suffers when the size of the indexed dataset becomes very big compared to the domain, or when the distribution of the items is skewed. In these cases, some inverted lists become very long and compromise the performance of query evaluation. This is due to the fact that the evaluation algorithms resolve to merge-joins between the lists. Huge collections of low cardinality set-values from a limited or skewed domain of items, appear often in practice. From the statistics provided by the US Food Marketing Institute [23], we infer that only in 2005, there have been almost 18 billion transactions (i.e., sets of products bought

at a time), in US supermarkets, with the average supermarket having 45k different products. Indexing such data with inverted files to provide efficient containment query evaluation (as part of market analysis tasks) would be the best solution available, but still the performance would not be satisfactory. The problem is further augmented by the fact that users usually pose queries involving the most frequent items in the dataset [8].

To compensate for this shortcoming, we propose a novel indexing scheme, the Ordered Inverted File (OIF). OIF first orders the set-values according to their items and then indexes them similarly to the classic inverted file. The ordering of the set-values confines the merge-join only to continuous subsets of the inverted lists that are relevant to the queries. Since these subsets are in fact ranges over the ordered set-values, OIF employees B-trees to index them and further prune the number of disk pages that need to be retrieved from the hard disk.

The primary goal of OIF is to reduce the I/O cost for containment queries. Our approach focuses on non-textual set-valued data, that become more and more apparent in practice, e.g., transactions from retail stores, web logs etc. This kind of data is characterized by skewed distributions and a a large ratio between the number of transactions and the size of the items' domain. We assume a context where the main memory is limited and the index cannot be memory resident. Many systems employing inverted files are systems dedicated to answering a single type of queries (e.g., superset in publish/subscribe systems) and they can afford using only memory resident indices. If there is sufficient memory, the basic cost in query answering is the CPU cost and specialized techniques like skip lists [35], or other memory resident structures [26] are used to reduce this cost. On the other hand, in this work we focus on containment queries under limited memory budgets. This is the case of a database that contains both the set-valued data and other information and has to respond to various types of queries. As our experiments demonstrate the I/O cost is dominant in evaluating containment queries, if the index is disk resident. We stress that the proposed approach is not a panacea for all kinds of settings and queries: therefore, we do not consider textual data sets that are traditional in IR and have very different characteristics; we do not consider specialized systems that can afford to have all the indices in main memory; we also do not consider approximate or ranking (i.e., similarity) queries.

In short, our contributions are:

- We propose a novel indexing scheme, the ordered inverted file (OIF), which outperforms the current state-of-the-art for containment queries. Our proposal is simple to implement and provides superior performance in all cases.
- We provide new evaluation algorithms for subset, set-equality and superset queries that take advantage of the proposed index.

- We show that the OIF performs reduced disk I/O operations compared to the classic inverted file and scales significantly better. Moreover, we test OIF's performance with an implementation that is built on the Berkeley DB embedded database, and we assess our proposal by extensive experiments on both real and synthetic data.

The rest of the paper is organized as follows: Chapter 2 provides the problem setting and the necessary background. In Chapter 3, we present the structure of the OIF index. In Chapter 4, we present the query evaluation algorithms and discuss their performance. Chapter 6 includes our experimental evaluation, and Chapter 7 places our contribution with respect to related works. Finally we conclude the paper in Chapter 8.

Chapter 2

Background

Consider a database D, where each record t has two fields: a unique key $t.id$ and a set-valued attribute, $t.s$. There are more than one ways to store such data. In the *object-relational* model, attributes are allowed to be set-valued, therefore we can store D, as a table with two attributes $t.id$ and $t.s$, as described above and depicted in (Fig. 2.1). In the pure relational model, set-valued attributes correspond to a set of tuples, therefore D is modeled as a table with two attributes id (which is no longer a key) and $item$, which takes a single value. For example, in this model, the first tuple of Fig. 2.1 would be represented as four tuples $(101, g)$, $(101, b)$, $(101, a)$, and $(101, d)$. Our method applies to both data organizations; for simplicity, in the rest of this paper we will assume the object-relational one. The active domain of $t.s$ is a finite set of values denoted as *vocabulary* I (i.e., the values $a, b, c, d, e, f, g, h, i, j$ for the database of Fig. 2.1).

2.1 Queries

In set containment queries, the user specifies a query predicate and a *query set* qs. The queries we are interested in are the following:

- *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, whose set-value is identical to 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 all contained* in the query set, i.e., $\{t \mid t \in D \wedge qs \supseteq t.s\}$.

As an example, assume that the data of Fig. 2.1 are the entries of a web log that trace the areas visited in a specific portal. Each record represents

id	*s*
101	{g, b, a, d}
102	{a, e, b}
103	{f, e, a, b}
104	{d, b, a}
105	{a, b, f, c}
106	{c, a}

id	*s*
107	{d, h}
108	{b, a, f}
109	{b, c}
110	{j, b, g}
111	{a, c, b }
112	{i, d}

id	*s*
113	{a}
114	{a, d}
115	{j, c, a}
116	{i, c}
117	{a, c, h}
118	{d, c}

Figure 2.1: Exemplary relation D

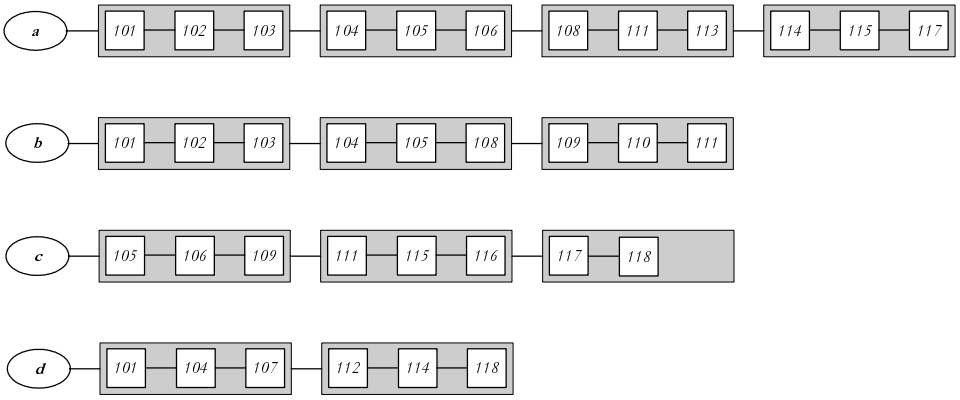

Figure 2.2: Partially shown IF for the example of Fig. 2.1.

a different user session and items in I (i.e., a, b, c, etc.) model URLs. The containment queries have intuitive meanings in all cases, e.g., "Which users limited their visit in the portal in the main and downloads sections?" (superset query).

2.2 Inverted files

The inverted file [25, 49] is composed by two main parts: (a) the *vocabulary table*, which contains all distinct items that appear in the database, and (b) one *inverted list* for each item, which includes references to the sets that contain the item. The inverted lists of four items (a, b, c, and d) from the database of Fig. 2.1 appear in Fig. 2.2. The gray boxes in Fig. 2.2 represent disk pages. The inverted lists can be very long for large databases, therefore it is natural to assume that they are stored in the secondary storage, while the vocabulary can fit in main memory. The latter is usually organized as an array, with a link from each entry to the inverted list, which contains references to all sets that include the respective item. Inverted lists are placed in contiguous regions in the disk, since querying requires to retrieve the whole lists that are linked to the query items [27].

id	Items
1	{a}
2	{a, b, c }
3	{a, b, c, f}
4	{a, b, d}
5	{a, b, d, g}
6	{a, b, f}

id	Items
7	{a, b, f, e}
8	{a, b, e}
9	{a, c}
10	{a, c, h}
11	{a, c, j}
12	{a, d}

id	Items
13	{b, c}
14	{b, g, j}
15	{c, d}
16	{c, i}
17	{d, i}
18	{d, h}

Figure 2.3: Example relation D with sorted ids

A subset query qs is evaluated by fetching the inverted lists of all items in qs and intersecting them. This computes the record-ids that contain *all* items in qs. For example, applying the subset query $qs = \{a, d\}$ returns $\{101, 104, 114\}$, which are indeed the only records in D containing both a and d. For processing equality and superset queries, the inverted file is extended, so that for each record-id in an inverted list, we also store the length (i.e., cardinality) l of the respective set [21]. An equality query qs is then processed in exactly the same way as a subset query, but records with cardinality different than $qs.l$ are directly pruned while traversing the lists. A superset query is processed by computing the union of the inverted lists for the qs-items (as opposed to their intersection for subset queries). While merging, we count the number of occurrences of each id in these lists. If for a record this number is equal to its length then we know that the record is a result to the superset query (since the record does not contain any items outside qs). For example, the superset query $qs = \{a, c\}$ returns records 106 and 113, since (i) these records appear in the inverted lists of either a or c and (ii) their cardinalities equal their occurrences in the inverted lists (e.g., 106 has two values and appears in both inverted lists).

Chapter 3

The Ordered Inverted File

Our proposal, the *ordered inverted file* (OIF) is an extension of the classic inverted file, based on the introduction of an ordering for the database items and records. Examples of possible item orderings include the ordering by frequency, alphanumeric value, etc. Later in the paper, we demonstrate that when containment queries are posed, this ordering allows the identification of specific areas in the inverted lists that contain potential answers to the queries. By coupling this property with a B-tree that organizes the lists as blocks in sequential disk pages, 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 B-tree, which organizes the access to all the parts of each inverted list. The search key of the B-tree is based on the value of the last record referenced in the corresponding block.

3.1 Ordering of the inverted lists

The ordering we adopt in this work for the records is based on the ordering of the items of the vocabulary I. Let the support $s()$ be a function that returns how many times an item appears in database D. Then, for any two items $o_i, o_j \in D$:

$$o_i <_D o_j = \begin{cases} true & if \quad s(o_i) > s(o_j) \\ true & if \quad s(o_i) = s(o_j) \wedge o_i <_a o_j \\ false & \text{otherwise} \end{cases} \tag{3.1}$$

where $<_a$ stands for alphabetic order. Equation 3.1 provides the definition of a *total* order operator $<_D$ based on frequency. Based on $<_D$, we define

the *sequence form*, sf, of a set-value.

Definition 1 *Given a set-value v from a database D, with n items we define as the sequence form of v, sequence $sf(v) = o_1, \ldots, o_n$ where $o_i <_D o_j$ for $i < j$.*

Consequently, the set-values are ordered *lexicographically*, after considering their sf and comparing their contents using the $<_D$ operator. The empty set comes first in this order, followed by the set containing the smallest item in the $<_D$ order. Following this lexicographic order, we can assign new ids to the records in D that can enhance search performance, as we discuss later. Fig. 2.3 shows the database of Fig. 2.1 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 $sf(t_1.s) \leq sf(t_2.s)$ in the lexicographic order. Depending on the requirements of the application, we can create these ids in two ways: (a) by using intermediate arrays that associate logical ids with physical links or (b) by placing the records themselves sequentially in the hard disk following the new order; in this case we can directly treat the physical links as ids. In the latter case the ids of Fig. 2.3 are enough to directly access the records, in the former the ids of Fig. 2.3 must be "translated" to physical addresses in order for a record to be retrieved. This is often the way simple inverted files work [16].

In the sequel, for simplicity, we drop the subscript D when referring to the order between items or sets. For example, $o_i < o_j$ implies $o_i <_D o_j$ and $t_i.s < t_j.s$, implies that the sequence form of set $t_i.s$ is lexicographically smaller than $sf(t_j.s)$, with respect to $<_D$. In addition, when we refer to a specific set-value, we list the items in it, in their $<_D$ order. For example, when referring to set $\{o_i, o_j\}$, it is implied that $o_i < o_j$.

3.2 Tagging for inverted lists

Knowing that records are referenced in lexicographic order (based on $<_D$) in the inverted lists can hint towards the contents of these records and whether they can contain results to a given containment query qs. Details on how containment queries are evaluated using the OIF will be provided in Chapter 4. To facilitate search, we consider the inverted list as a sequence of blocks, and we employ a *tagging* mechanism that provides the lower bound for each such block. Each block is associated with a tag, which is the sf of the last record that is referenced in the block. This way, we can avoid fetching the next block from the hard disk, if we can infer that it is unnecessary. For example, consider an inverted list, which contains a block with record ids $7, 8, 9$ of Fig. 2.3. The tag of this block is $\{a, c\}$, i.e., the contents of the last record (9) in the block.

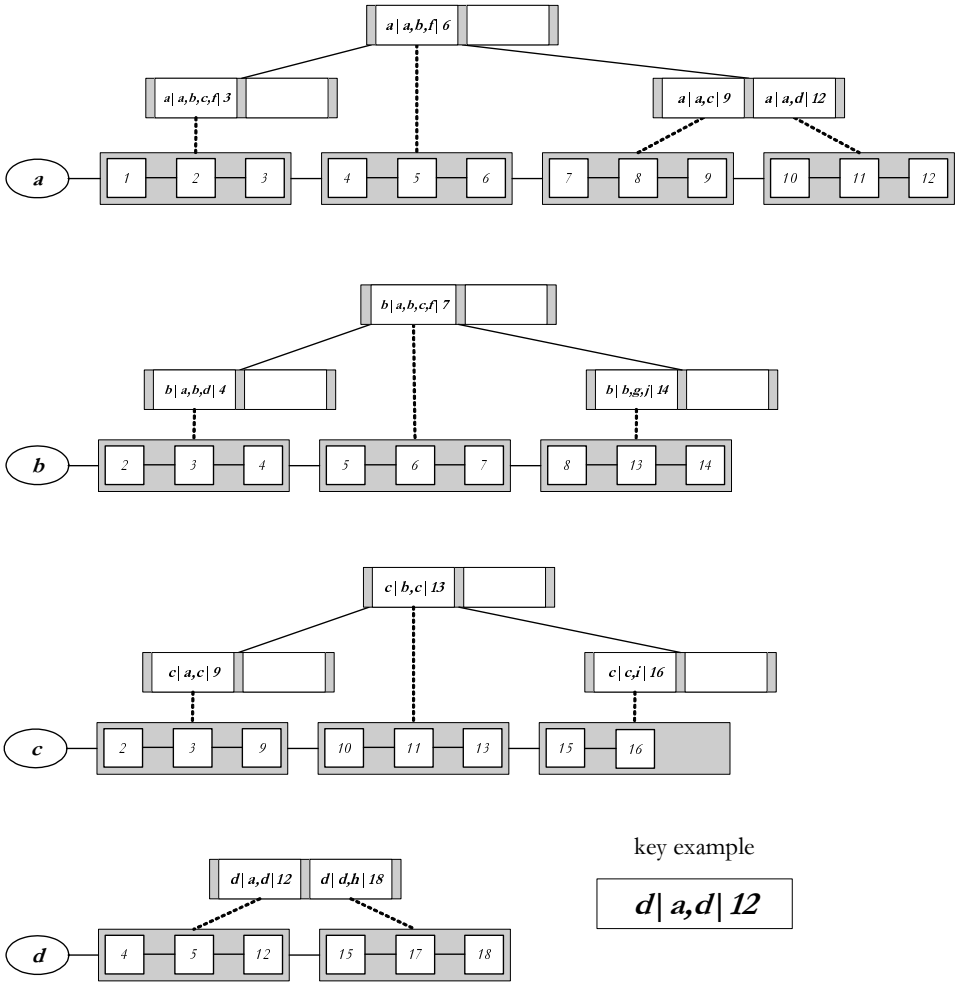

Figure 3.1: Partially shown OIF for data of Fig. 2.3.

3.3 B-tree indexing for inverted lists

In the classic inverted file the list is simply positioned in sequential disk pages (in the ideal case). Since the tagging mechanism allows for pruning parts of the inverted list during query evaluation, the OIF provides direct access to intermediate points of the inverted list. To this end, each list is broken in blocks that are organized in a B-tree. The B-tree stores blocks by exploiting the associated tags in the key definition. Each entry in the B-tree has four parts: (a) the item that is associated with the inverted list, (b) the tag and (c) the id of the last record of the block, which form the *key* and (d) the associated block. The record-id serves two purposes: (a) it guarantees that each key will be unique and (b) we use it to speed up query evaluation

(we give details in Chapter 4). Constructing the key in the previous way guarantees that all the blocks that belong to one list will be sequential entries in the B-tree. A part of the OIF for the database of Fig. 2.1 is depicted in Fig. 3.1. We do not place the block inside the graphical node, but instead we trace the link between the key and the block by a dotted line for reasons of readability. Moreover, we depict each inverted list indexed by a different B-tree, instead of having one single B-tree for all blocks as in the actual implementation. In the actual implementation we store all blocks in a single B-tree (since we use a single relation in the BerkeleyDB). Using the item id as a prefix to the key guarantees a unique key for each block.

Each block of the inverted list introduces only one entry to the B-tree. The size of each key depends on the average cardinality n of the set-values. Assuming each item and record-id occupies k bytes, each key takes $(n+2)\times k$ bytes. This size can be reduced, by (i) compressing the inverted lists and reducing the required number of blocks, and (ii) considering prefixes of the ordered set-values used as tags.

3.4 Metadata

An interesting consequence of assigning ids to records according to the order we introduced in Equation 3.1, is that the combinations of the most frequent items of each record define a contiguous region over the id space. For example, all the records whose smallest (i.e., most frequent) item is o are assigned successive ids in the id space. It is this item, o, that plays the most important role in the placement of the record in the total order of D. Based on the previous discussion, we can easily show the following theorem:

Theorem 1 *For each item o, we can identify a contiguous region $[l, u]$ in the id space which corresponds to only and all records whose smallest item is o.*

This theorem allows us to reduce the size of the inverted lists; instead of indexing all the ids of the records whose smallest item is o in the inverted list of o, we simply store the boundaries $[l, u]$ of the region that contains them. The smallest the item o is in the total order of I, the greater the effect of keeping only the boundaries $[l, u]$ will be; if $o = a$ then all the records who contain a will reside in $[1, u_a]$, since there is no item that is smaller than a. If $o = b$, then in $[l_b, u_b]$ the ids of all records that contain b and not a will be included, since it is only item a that is smaller than b. Ordering the items of I as we did in Equation 3.1, i.e., according to their frequency of appearance, maximizes the gains of keeping only the $[l, u]$ boundaries instead of all the ids. We store all these regions in a *metadata* table, which we consult during runtime.

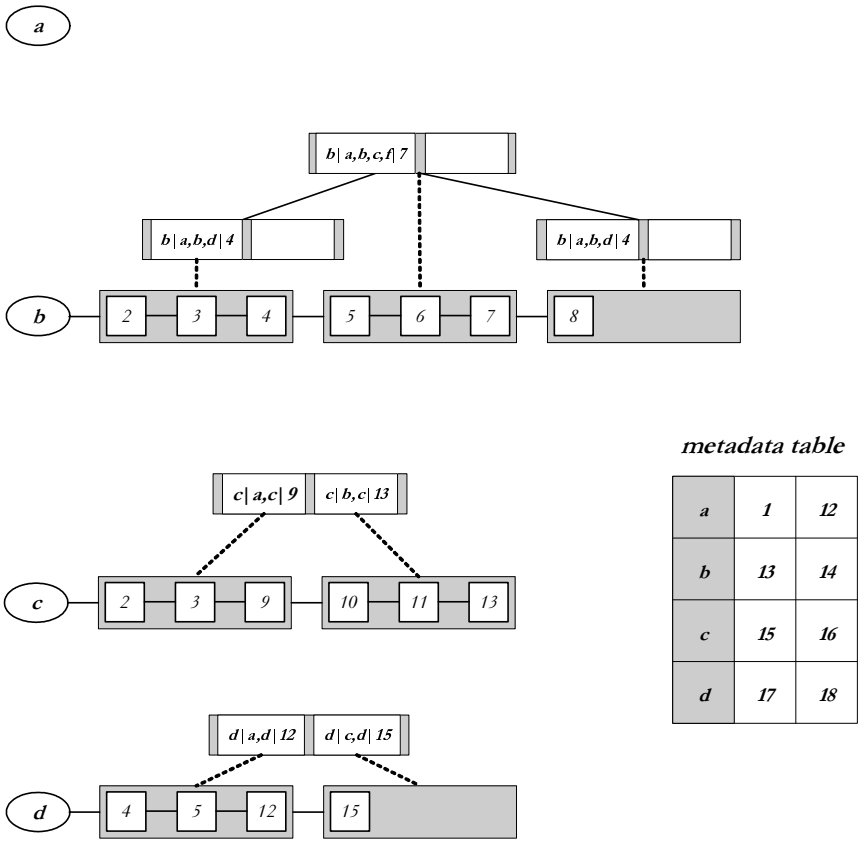

Figure 3.2: Using a metadata table can reduce the OIF of Fig. 3.1

The ordering also has one more interesting consequence. If $[l, u]$ is the contiguous region that contains records whose smallest item is o, then no record with id greater than u will contain o. If o is the smallest item of the records that appear in $[l, u]$, all records which appear after u will have a smallest item $o' > o$, else they should have been contained in $[1, u]$. Moreover, since the $[l, u]$ region lies at the end of the $[1, u]$ range, $[l, u]$ always describes the suffix of o's inverted list in the OIF. By bookkeeping this suffix as a range $[l, u]$ in the metadata table, we simply cut the lists shorter. An example of a more compact version of the OIF, which uses a metadata table, for the database of Figure 2.3 is depicted in Figure 3.2. As it will become clear in Chapter 4 smaller lists lead to faster query evaluation. For instance, to answer the subset query $\{a, b\}$ using the OIF of Figure 3.2, it suffices to intersect the inverted list of the least frequent item (b) with the metadata range of a, which is cheaper than the intersection of the two lists of the OIF depicted in Figure 3.1.

3.5 Compression

The most common technique for reducing I/O accesses for the simple inverted file is compression. Several compression techniques have been proposed in literature [32, 38]. Their main idea is to code the contents of the inverted lists in some variable bit or variable byte encoding to reduce storage requirements. To further enhance the effect of the encoding, they avoid storing the actual record ids in each inverted list, but instead they store the *d*-gaps between them; the *d*-gaps are the differences between sequential ids. For example the inverted list of item d as shown is Figure 3.1 is $\{2, 5, 12, 15, 17, 18\}$; the *d*-gaps would be $\{2, 3, 7, 3, 2, 1\}$. The benefit from this approach is that the inverted list contains smaller values, thus the variable size encodings are more efficient. In practice, compression techniques can reduce the size of the inverted file, down to 30% of the original size, which can be even smaller than the database size [48]. These techniques are directly applicable on the OIF and they have an additional benefit due to the order we adopt in Equation 3.1. Instead of having in each list ids from any part of the database, we only have ids ranging from $[1, u]$ or $[1, l]$ if we use the metadata table, thus their average *d*-gaps are smaller. In our implementation we adopt a byte-wise compression scheme [44], due to its reduced CPU cost in the decompression phase.

Chapter 4

Query evaluation

In simple inverted files, where the ids of the records have no special meaning, the answers to an arbitrary query are usually scattered throughout the inverted lists of the involved items. The evaluation of set-containment queries over inverted files, is reduced to merge joins between the inverted lists, as discussed in Chapter 2. Thus, in order to retrieve the query result we usually have to scan the entire lists. There has been extensive work on list intersection algorithms, that is applicable to inverted files [6, 39, 42]. The focus in these works lies in reducing the CPU cost, since they are mostly aimed at specialized systems, which answer few types of queries and can afford to have all lists in main memory. In this chapter we do not try to propose novel list intersection algorithms, but rather proof-of-concept algorithms that will demonstrate how OIF can significantly reduce the I/O cost in query evaluation. Most list intersection algorithms that work for the inverted file can also exploit OIF to retrieve the lists from the disk, without significant changes. Such a scenario does not reduce the effectiveness of OIF since it still reduces the I/O by retrieving only the part of the list that is related with the query. Moreover, OIF results to smaller lists due to the use of the metadata. The basic aim of our query evaluation algorithms is to offer a fair basis of comparison between the inverted file and the OIF. The experiments of Chapter 6 show that even if the CPU cost of a query posed against an inverted file is 0, it would still be slower than the OIF due to its increased I/O.

The OIF reduces the I/O by having the records ordered in the inverted lists as we proposed in Chapter 3. The ordering allows the identification of a range of record-ids that contains the answer to each query. We call this the *Range of Interest* (*RoI*) of the query. The knowledge of the theoretic bounds of a query's *RoI* allows the effective utilization of the B-tree; instead of accessing the complete inverted lists, we use the tree to locate and access only the blocks that contain candidate results.

The algorithms for the evaluation of the subset, set-equality and superset

queries against a database indexed by an ordered inverted file have two basic steps:

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

The most significant difference from the evaluation of the same queries against classic inverted files lies at the first step. Using the RoI, the part of the inverted lists that is examined is significantly pruned. Query evaluation also benefits from the existence of the metadata, which reduce the size of the lists and consequently the amount of data to be joined. Finally, the join exploits the direct access to different blocks provided by the B-tree. During query evaluation the candidate solutions might be limited to a region smaller than the RoI. Using the B-tree we can access only this region, further increasing the pruning done by the RoI.

The bounds of the RoI can be computed based only on qs and the type of the query predicate. Having identified them, we know the broadest possible region of the search space that contains the answer. Using the B-tree we can trace the sequence of blocks whose tags cover the RoI, without accessing the whole inverted list. The first block of this sequence is the first block 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 the RoI. For reasons of simplicity we overload the notation of RoI to also denote the sequence of blocks that covers the actual RoI in the description of the algorithms.

In the rest of this chapter, we present the evaluation algorithms for each query predicate and we define the RoI for each case. Moreover, we show that examining the RoI is adequate to trace the entire answer to each query. The proofs of the theorems are straightforward in all cases so we refer the interested reader to [40] for further details.

4.1 Subset queries

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

Definition 2 (**RoI$_{\mathbf{sub}}$**) *Given a subset query* $qs = \{o_{q_1}, \ldots, o_{q_n}\}$, $o_{q_1} < \cdots < o_{q_n}$ *containing items from a domain* $I = \{o_1, \ldots, o_N\}$ *with* $o_1 < o_2 < \cdots < o_N$, *the Range of Interest* RoI_{sub} *is a region with the lower bound being set to* $\{o_1, o_2, \ldots, o_{q_n)}\}$ *and the upper bound set to* $\{o_{q_1}, o_{q_2}, \ldots, o_{q_n}, o_N\}$.

Theorem 2 *It is sufficient to search for records within* RoI_{sub} *to answer a subset query.*

Assume, for example, that $I = \{a, b, \ldots, j\}$ and we would like to retrieve all the records that contain $qs = \{b, c\}$ from the relation of Fig. 2.1. Then,

$RoI_{sub} = [(a, b, c), (b, c, j)]$. Any record t with $t.s < \{a, b, c\}$ does not contain c and any record t' with $t'.s > \{b, c, j\}$ lacks b or c or both.

The algorithm for evaluating subset queries is depicted in pseudocode in Algorithm 1. The evaluation starts by joining the smallest inverted lists, which belong to the greatest items (i.e., least frequent ones). This is done in order to detect possible void answers faster and to avoid having a large number of candidate solutions. At each step, the algorithm performs 2-way merge intersection join between the candidates and the current list. Each join can only reduce the number of candidates, since valid solutions need to appear in all the inverted lists. Knowing that during query evaluation we are only going to dismiss candidate solutions and not to discover any new ones allows for further pruning. We can progressively limit the range of the list that we are going to examine (initially, equal to RoI_{sub}) considering the minimum and the maximum candidate ids. For example, if our candidates are $\{4, 6, 9, 12\}$ then we only need to examine the part of the remaining inverted lists that contains ids in the $[4, 12]$ range. If after the next intersection join the remaining candidates are $\{4, 6, 9\}$ we can further limit this range to $[4, 9]$. The candidates are kept as in a sparse set representation in memory (i.e., as a bitset or list depending on the number of candidates). Intersections are then performed as bitwise joins or merge joins.

Algorithm 1 Subset Query Evaluation

subset$(qs = \{o_{q_1}, \ldots, o_{q_n}\}, o_{q_1} < \ldots < o_{q_n})$
1: determine RoI_{sub}
2: retrieve $candidates = l_n \cap RoI_{Sub}$ ▷ we insert to candidates the part of l_n that falls in RoI_{sub}
3: **for all** $o_i \in (qs \setminus o_{q_n})$ **do** ▷ we take items from qs in reverse order, i.e. $o_{q_{n-1}}, \ldots, o_{q_1}$
4: $range = RoI_{sub}$
5: **for all** (blocks b_i in $range$) **do** ▷ we access blocks b_i from the list of o_i sequentially.
6: **for all** postings c_i in $candidates$ **do**
7: **while** posting p from b_i $p < c_i$ **do**
8: get next posting p from b_i
9: **if** $p \neq c_i$ **then**
10: remove c_i from $candidates$
11: **if** $o_i == o_1$ **then**
12: **for all** remaining postings c_i in $candidates$ **do**
13: **if** c_i not in metadata bounds **then**
14: remove c_i from $candidates$
15: $range = [lid_c, uid_c]$ ▷ lid_c is the id of the first item in $candidates$ and uid_c is the id of the last
16: return $candidates$

The direct effect of using the metadata in the evaluation of the subset queries is basically limited to the smallest (i.e., the most frequent) item (lines 11-14 of Alg. 1). For all other items, the part of the list that is described solely by the metadata is always out of the RoI. For example, the metadata range of item c is irrelevant to query $qs = \{b, c\}$, because this range contains records that have c as their smallest item (and therefore cannot contain b). Therefore, the evaluation of a subset query $qs = \{o_{q_1}, \ldots, o_{q_n}\}$ using the metadata, follows the lines of Algorithm 1, however, the inverted list of o_{q_1}

needs not be accessed; it suffices to intersect the current candidate set with the metadata range of o_{q_1}.

The worst case for the OIF is to examine the whole affected lists, if the RoI_{sub} fails to prune any block. To estimate the part of each inverted list that is accessed during subset evaluation, one has to estimate the placement of the RoI_{sub} in the ordered records. The lower bound of the RoI_{sub} is always positioned very near to the start of the list, so the pruning depends mainly in the placement for the upper bound. For a $qs = o_{q_1}, \ldots, o_{q_n}$ this bound is $RoI_u = o_{q_1}, \ldots, o_{q_n}, o_N$, where o_N is the last item of I. Since the order of records is lexicographical the most important term in the formula is the first. An immediate consequence of this is that the greater the least item of qs is, the worse the RoI_u will be, and thus the pruning result from the RoI will be limited. On the other hand, in a skewed distribution, an item that is big in I will be infrequent and its inverted list small, thus the subset query evaluation for a qs where all the items are big in I will be computationally cheap (if the smallest item is big in I, then all the record's items will be big in I). In other words, the greater the inverted lists that must be accessed during query evaluation are, the greater the pruning provided by the RoI, will be.

4.2 Equality queries

While in the classic inverted file equality queries have similar evaluation cost to subset queries, in the case of the OIF they are significantly less expensive to compute. 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 3 ($\mathbf{RoI_{eq}}$) *Given an equality query with a query set qs, RoI_{eq} is a continuous region with lower and upper bound the query set qs.*

Intuitively, to evaluate an equality query, we trace the blocks in each inverted list that contain the answer via the B-tree and merge-join them. The evaluation algorithm is virtually the same with the subset evaluation algorithm depicted in Figure 1. The basic difference is that now we use the RoI_{eq} and that we filter records according to their set-length, which is stored in the inverted lists; only records that have length equal to the query cardinality are considered in the merge join.

When the data are indexed by a classic 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 as an additional filter. The RoI_{eq} practically defines a single block in each list, i.e., $|qs|$ entries in the B-tree. The only exception to this happens when there exist enough duplicates of the qs that do not

fit in a single block, and take up several blocks on the index. Moreover the list of the smallest item in the query needs not be accessed at all, since all the valid answers must lie inside the bounds traced in the metadata table for this item. As a result, the evaluation of the equality queries with OIF depends linearly on the size of the query set $|qs|$ and only logarithmically to the size of the inverted lists and consequently to the size of D. The logarithmic dependence is due to the traversal of the B-tree, which takes place in the process of deciding the RoI_{eq} of the query. Thus the cost of evaluating an equality query is $O(|qs|log_b(\frac{|D|}{b_c}))$, where b is the order of the B-tree and b_c is the number of postings that fit in a block.

4.3 Superset queries

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). A superset query is equivalent to $2^{|qs|}$ equality queries. Defining the Range of Interest as $2^{|qs|}$ distinct points is not efficient, thus we group them in larger areas, based on the first (i.e., most significant in the ordering) item of each combination. Unlike the case of the subset and equality queries, the Range of Interest for a superset query is different at each inverted list. This follows from the fact that each inverted list contributes to a partly different set of solutions (i.e., all the solutions that contain the associated item), since in superset it is not required that all the qs items are present in all lists, as in the previous queries.

Definition 4 (**RoI$_{\mathbf{sup}}$**) *Given a query set $qs = \{o_{q_1}, \ldots, o_{q_n}\}$, where $o_{q_1} < \cdots < o_{q_n}$ the RoI_{sup} is defined as*

$$\begin{aligned}
RoI_{sup-o_{q_1}} &= [(o_{q_1}), (o_{q_1}, o_{q_n})] \\
RoI_{sup-o_{q_2}} &= [(o_{q_1}, o_{q_2}), (o_{q_1}, o_{q_2}, o_{q_n})], [(o_{q_2}), (o_{q_2}, o_{q_n})] \\
&\vdots \\
RoI_{sup-o_{q_n}} &= [(o_{q_1}, \ldots, o_{q_n}), (o_{q_1}, o_{q_n})], \ldots, [(o_{q_n}), (o_{q_n})]
\end{aligned}$$

Intuitively, in the list of the smallest item o_{q_1} of qs we identify the region of all records whose smallest item is o_{q_1}. In the list of the second smaller item of qs, o_{q_2} we identify two regions of interest: one that contains the records where the smallest item is o_{q_1} and one with records whose smallest item is o_{q_2}. For o_{q_3}, we identify three such regions and so on. The benefit of this region definition is that the last region always falls in the range of ids stored in the metadata table and not in the inverted lists, thus it is a lot cheaper

to verify if an id is included or not in it.[1] In Figure 4.1, the RoI_{sup} for the case of $qs = \{a, c, f\}$ is depicted. To make it more easily interpretable we represent the records with all their items (instead of their ids) and we depict all the bounds of the regions like existing records.

Algorithm 2 Superset Query Evaluation

superset$(qs = \{o_{q_1}, \ldots, o_{q_n}\}, o_{q_1} < \ldots < o_{q_n})$
1: determine RoI_{sup} ▷ we use RoI^j_{sup-i} for the j-th RoI of item i
2: retrieve $candidates = l_n \cap (RoI^i_{sup-n} \cup \ldots \cup RoI^n_{sup-n}) \wedge length \leq |qs|$ ▷ we retrieve the part of the list l_n of o_n that falls in RoI_{sup-n} and have record length less or equal to $|qs|$
3: $results = \emptyset$
4: **for all** $o_i \in (qs \setminus o_n)$ **do** ▷ we take items from qs in reverse order, i.e. $o_{n-1}, \ldots, o_{n-1}$
5: $range = RoI^1_{sup-i}$
6: **for all** (blocks b in $range$) **do** ▷ we access blocks b from the list of o_i sequentially.
7: **for all** (postings p in b) **do**
8: get next c from $candidates$
9: **while** $(c < p)$ **do**
10: **if** $(c.length - c.found > |qs| - i)$ **then**
11: remove c from $candidates$
12: get next c from $candidates$
13: **if** $(p > c)$ **then**
14: **if** $(p.length \leq |qs| - i)$ **then**
15: insert p in candidates
16: **else**
17: $c.found{+}{+}$
18: **if** $(c.length = c.found)$ **then**
19: remove c from $candidates$
20: insert c in $results$
21: $range = RoI^x_{sup-i}$ ▷ RoI^x_{sup-i} is the next RoI of i which is not covered by the previously examined block b
22: **if** $(range = RoI^i_{sup-i})$ **then** ▷ if it is the last RoI for i
23: insert all the ids that lie in $[l, u_1]$ to $results$
24: increase the $found$ counter for all candidates that appear in $[u_1, u]$
25: return $results$

The algorithm for evaluating superset queries with the OIF index is depicted in Figure 2. The evaluation of superset queries is more expensive than the evaluation of subset and equality queries, because candidates do not get disqualified just for not appearing in one inverted list. To decide that a certain record is a valid solution, it must appear in the inverted lists of qs as many times as its length, i.e., all its items must appear in qs. We can discard a candidate only if it has missed enough matches in the examined inverted lists, and safely decide that it contains additional items to those

[1] A subtle problem arises from the fact that we do not trace the length of the sets in the metadata. This problem is actually limited only to records of size 1. Since the regions stored in the metadata have no overlap, it is certain that if a record contains more than 1 items, a posting containing its id and size should appear in at least one more list. If this id qualifies as a solution for the superset query, then the evaluation algorithm must have examined it and it must know its set cardinality. Finally, it is easy to solve the problem of records that contain only one item by adding one more field in the metadata table; the upper bound u_1 of the region $[l, u_1]$ which corresponds to ids of the records that contain only one item. This region will always lie in the beginning of $[l, u]$.

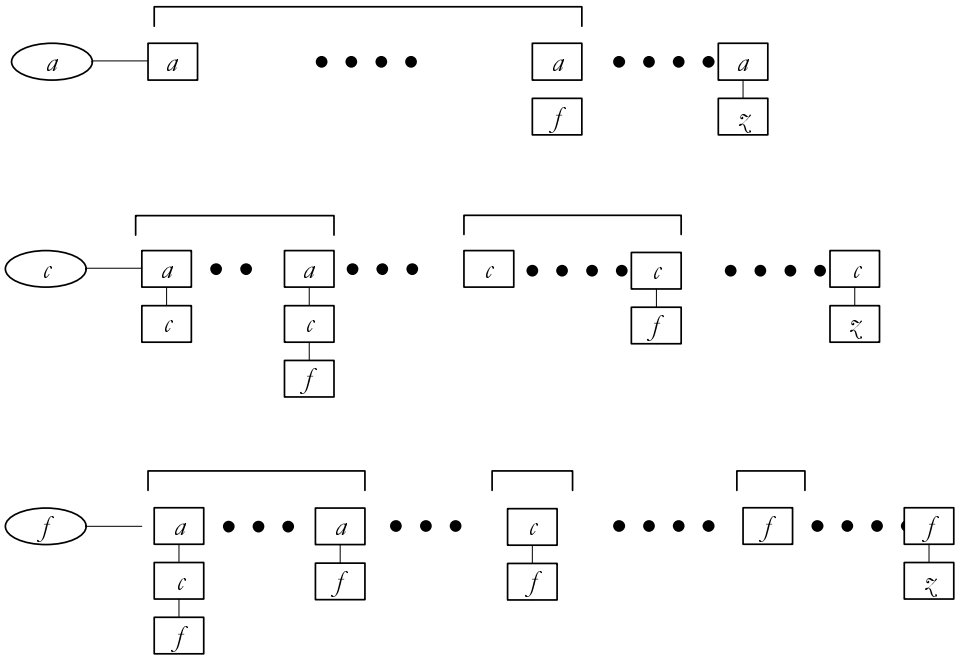

Figure 4.1: The ranges of interest for the superset query $\{a, c, f\}$.

that appear in qs. To this end, for each candidate c we keep the additional information $c.found$, marking the number of times we encountered c in the examined inverted lists. After examining each list, we check if the total items of c that have not yet been found (i.e., $c.length - c.found$) are more than the remaining unexamined inverted lists. If this holds, then we can safely discard c as a candidate, since it contains at least one item that does not appear in qs. Unlike the case of subset and equality queries, where each successive merge-join only disqualifies candidates, here, joining the candidates with additional inverted lists can result in an increase to the candidate members. Actually, all entries that fall inside the RoI_{sup} should be inserted in the candidates, except the case where we can conclude from their length that they cannot be valid solutions. When the algorithm has completed the join of the lists, it has to examine the ids from the metadata table. Again, it increases the $found$ counters of existing candidates that were found. No new candidates will be inserted from this region, except those having length one. Those containing multiple items should have already been inserted to candidates, since the additional items can only be greater than the one currently examined. Finally, before asking for the blocks that are contained in the next RoI_{sup} of the same inverted list, the algorithm checks if this RoI_{sup} is not already included in the previously retrieved block. Such might be a common case for small inverted lists.

The basic steps of the algorithm for evaluating superset queries are common both for classic inverted files and for ordered inverted files. Their performance is differentiated due to the ability of the OIF to limit its search to a limited part of the inverted list and its ability to exploit the metadata table. A coarse bound for the RoI's in each inverted list is the range between (o_{q_1}, o_{q_i}) and (o_{q_i}, o_{q_n}), where o_{q_i} is the item that is linked to the inverted list, o_{q_1} is the first item of the query set and o_n the last. As a general result we have two conclusions: (a) the closer the items are placed in the order of I, the more selective the RoI will be, and (b) the smaller the o_1 is, the smaller the RoI will be. Moreover, due to the usage of the metadata, the inverted lists stop before the appearance of the first record whose smallest item is o_{q_i}. The remaining records are examined by using the bounds $[u_1, u]$ for item o_{q_i} from the metadata table (see lines 23–25).

Chapter 5

Maintenance

The main difference between updating the OIF and the classic inverted file lies at the need to sort the data in order to provide new ids. This extra cost in the update procedure makes updating the OIF a more costly procedure than updating the classic inverted file, but still, the difference is not such that limits the applicability of OIF.

Inverted files are usually maintained mainly by batch, offline updates. A requirement for the inverted file to work efficiently is to store the inverted lists in a contiguous way on the secondary storage. This requirement renders impossible the naive solution to simply add the new entries at the end. If it is necessary to index the new records immediately when they come, we construct a second, small, memory-resident inverted file and index them there. During query evaluation we need to access both the disk resident and the memory based one in order to provide a complete answer. There are three strategies to update the disk-resident index: (a) to rebuilt it from scratch using both the old and the new data, (b) to merge it with the memory resident one and (c) to lazily update the inverted lists when they are retrieved from the disk during query evaluation. In the last case the retrieved list is merged with the memory resident one and the result in written to disk as a new list. A very helpful survey on update techniques for inverted files can be found at [47]. If the items' domain I is limited, the most suitable solution is the second one, since even small amounts of new data will probably affect all lists.

The main steps of the update algorithm for the OIF, are presented in Algorithm 3. The difference in the update cost compared to the classic inverted file comes mainly from three factors: (a) the need to merge-sort the old data with the new data in order to decide the new ordering for the ids, (b) the sorting of the updates (the main database is already sorted) and (c) the increased computational cost in the merging of the indices. Theoretically, an advantage of the inverted file is that only those lists that are affected by a batch update will be retrieved, merged, and re-written to

Algorithm 3 OIF update algorithm

update(old database: D, new database D_n, updates: U, index of D: OIF_D, index of U: IF_U)
1: initialize $DBmap$ ▷ $DBmap$ holds the mappings from old ids to new ones, for the records of D_n
2: initialize $Updmap$ ▷ $Updmap$ holds the mappings from old ids to new ones, for the records of U
3: sort the records of U
4: merge D and U ▷ $DBmap$ and $Umap$ are populated
5: **for all** (inverted list l of OIF_D) **do**
6: merge l with the respective list of IF_U ▷ old ids are translated to new ones via $DBmap$ and $Updmap$
7: write the new list

disk. OIF, on the other hand, must always reconstruct all its inverted lists. Practically, in the settings where the items domain is small compared to the data size, even for small updates most of the lists (or at least the largest ones which correspond to the most frequent items) are always affected by the updates. One advantage for the OIF is that its inverted lists are smaller due to the existence of the metadata table. Note that even in the case of the OIF, we keep the updates batch U indexed by a classic inverted file, since the index will be small and memory resident. Keeping the OIF updated is more costly than keeping a classic inverted file updated, but the fact that even classic inverted files need to be read from the disk, merged with the memory based index and re-written, makes both updating procedures to run in $O(|D|)$. The classic inverted file is updated in $O(|D|+|U|)$ while the OIF in $O(|D|+|U|+|U|\lg|U|)$ time (the OIF additionally requires sorting the updates which adds the $|U|\lg|U|$ factor).

In the update procedure, we assume that the ordering in I remains constant and that the initial database does not need re-ordering. We believe this is a reasonable assumption; even if the actual order, which is based on the frequency of appearance of the items, changes, the query evaluation algorithms will work correctly as long as we are consistent in both the ordering of the database and of the updates. Moreover, the definition we adopted for the order is a heuristic aimed at improving the compression factor and mostly at maximizing the usage of the metadata table. Small changes, that do not drastically change the ordering (e.g., some of the most frequent items to be considered in-frequent after the updates) will not cause significant performance changes. In most application areas like web logs, transactions from retails stores, etc., the appearance frequencies for the different items usually remain relatively stable over large update intervals.

Chapter 6

Experimental Evaluation

We evaluated the performance of the proposed OIF index by comparing it to the state-of-the-art for containment queries, the inverted file (IF). We implemented the two indexes in C++, using Berkeley DB [1] as our storage engine. Berkeley DB supports relations that have two fields: key and data. There is no type restriction for any of the fields and they can be any binary object. In our implementations each block is stored at the data field of the relation. In the implementation of the OIF, we split the inverted lists into blocks, which are stored as independent entries in the relation, with a B+-tree primary organization. Each block is associated with a key that is formed by combining the item that is associated with the list and the content of the last record (i.e., the tag), together with its id. Each inverted list is populated by postings which are comprised by the id and the length of the records. The ids are represented as series of d-gaps compressed by a v-byte compression. The same compression is used for the lengths of the records.

For the IF, we use the most efficient implementation scheme reported [30]: each tuple has as key value an item o from I and as data value the whole inverted list that is associated with o. The relation is indexed by a hash over the key values. The postings are compressed exactly as in the case of the OIF. Note that Berkeley DB always retrieves the whole tuple, i.e. there is no way to retrieve a part of the inverted list.

6.1 Datasets

We evaluated the OIF using two real datasets from UCI KDD repository [22]. 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

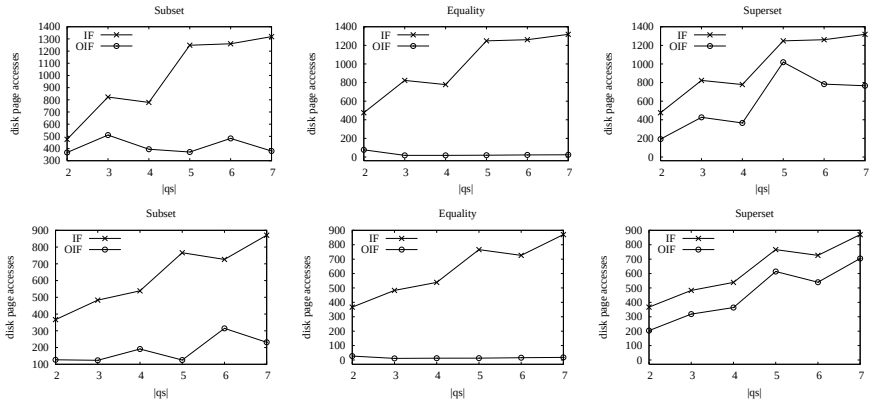

Figure 6.1: Average performance of queries on real datasets - First row *msweb* data and second row *msnbc* data

on a large database, we replicated the dataset 10 times. This replication is meaningful, since it simply simulates a 10-week log from the web portal. The second dataset, denoted *msnbc* is again a log of users behavior at another web portal, *msnbc.com*, containing 990K records. 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 cardinality of a set-value is 5.7.

To assess the performance impact of several statistical properties, we used synthetic data containing set-values with length varying from 2 to 20. We considered datasets of 1M, 5M, 10M and 50M set-values containing items from vocabularies of sizes 500, 2K and 8K. The frequency of items in the set-values is a moderately skewed Zipfian distribution of order 0.8. Unless explicitly stated we use as default parameters a domain of size 2K and 10M records with a distribution of order 0.8.

6.2 Queries

As in other approaches [21], 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 set-values, 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 the superset queries can surpass 3% for large $|qs|$ on the real data. To provide representative results we created 10 queries of each size and type.

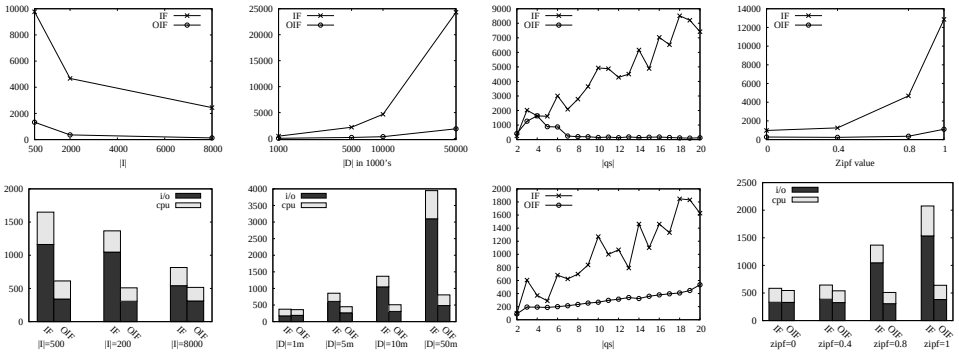

Figure 6.2: Average performance of subset queries on synthetic datasets - First row shows disk page accesses and second row time in msecs

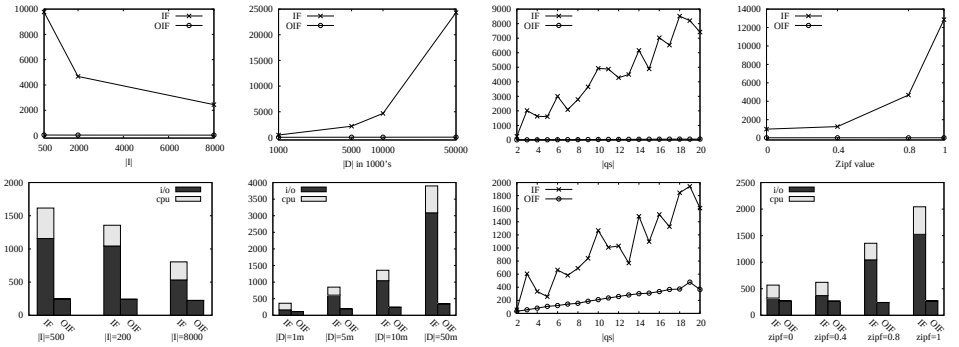

Figure 6.3: Average performance of equality queries on synthetic datasets - First row shows disk page accesses and second row time in msecs

6.3 Performance evaluation

In our experiments, we primarily evaluate the I/O cost. For this reason, we set up the database cache to the minimum (32K) and we circumvent the operating system cache. This leaves only the effect of the hard disk cache, which is limited. We minimize the effects of caching to observe more accurately how the index would scale for larger datasets, when given limited cache size. We would like to stress that the important factor is the percentage of available memory to the underlying database size: in realistic situations where the DBMS will have to simultaneously support multiple queries, the memory assigned per query is bounded. Therefore, our approach is intentionally tailored and tested to sustain bounded -in fact, small- memory budgets. We trace the actual disk page accesses, reported as cache misses by the database. Berkeley DB does not report separately sequential and random disk page accesses. The query evaluation algorithms need a few random disk page accesses to trace the beginning of the list (or of the *RoI*

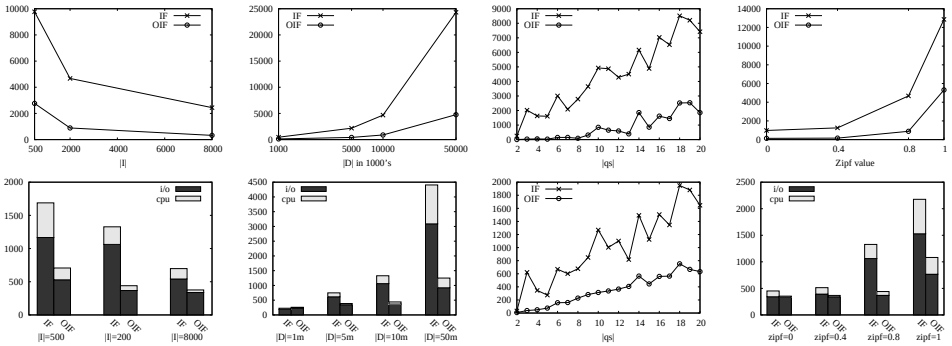

Figure 6.4: Average performance of superset queries on synthetic datasets - First row shows disk page accesses and second row time in msecs

in the case of the OIF) in the disk, and then they retrieve sequential disk pages from the inverted list. The random disk page accesses when using the OIF are more frequent than in the case of the IF, due to the B-tree indexing and the large number of keys. Still, our experiments show that this overhead is proportional to the query size and not to the database or the domain size and its effect is quite limited. To better understand how sequential and random accesses affect the IF and the OIF we present the query evaluation time and we distinguish it in CPU and I/O time by repeatedly counting clock ticks and total time in an system where no other operation runs. We present runtimes only for the synthetic datasets which are large enough not to be significantly distorted by caching and prefetching operations by the hard disk.

Subset. The OIF outperforms the IF both for real and for synthetic data. Moreover, it exhibits two very good properties (see Fig. 6.2). As the length of the query set grows and thus, queries have fewer answers, it manages to use the B-tree efficiently and detect fast the areas of interest in each list. This way, its cost drops, unlike the case of the IF, which suffers when it has to examine many inverted lists and cannot benefit from the small selectivity. Another good property of the OIF is that it scales well as the database D grows. This is again attributed to the B-tree and the limited RoI. When the size of the database and consequently the size of the lists is small, the random access I/O nullifies the advantages of the OIF and the I/O cost is similar to that of the IF for the smallest dataset of 1M records. As the size of the database grows, the gains of avoiding large amounts of sequential disc accesses grow evident. As we can see the I/O time grows as the query size grows, whereas the disk page accesses remain fairly stable. This reflects the fact that the OIF needs more random accesses for detecting the RoI in list, that the IF needs for detecting a list. So when the number of lists increases

the I/O time increases too due to the random accesses needed to find the *RoI* inside each list.

Equality. As expected from the complexity analysis, the cost of the OIF is practically constant (see Fig. 6.3), since it can access directly the small set of disk pages where the possible answer lies. On the other hand, the IF does not have the opportunity to prune ids using *RoI* and, as experiments on both real and synthetic data confirm, it has significantly higher cost. Finally, the ability of the OIF to prune the candidates very effectively, results in unnoticeable CPU cost.

Superset. Superset allows less pruning of the lists compared to the other queries, but still the OIF systematically outperforms the IF in both real (Fig. 6.1) and synthetic data (Fig. 6.4) for all domain, database, and query-set sizes. In the real data sets, it is clear that the benefits from the OIF are not as drastic as in the two other cases; this is due to the fact that since the databases and the vocabularies are rather small, the query-set size (and thus, the number of examined lists) is significant compared to the vocabulary. The percentage of pages examined by the OIF is still significantly lower than in the case of the IF, but the OIF is forced to do more random I/O that in other queries. This is reflected on the I/O times. The random I/O accesses are again proportional to the query size, and not to the domain or the database size. The pruning power of the OIF is reflected in the smaller CPU cost, which is a result of producing less intermediate results and of the exploitation of the metadata table. The benefits are more clear in Fig. 6.4 if one looks at the way the IF and the OIF scale with respect to the database size. Similar observations can be made as qs increases.

For all query types, it is noteworthy that the OIF is more robust to the *skewness* of the data compared to the IF. While in uniform distributions (zipfian order = 0) the two structures perform almost equally well (with the IF having a slight advantage in the superset queries), as the skew increases, the performance of the IF quickly drops by an order of magnitude for subset and equality queries and by 25-30% for superset queries.

6.4 Space overhead

As explained in Chapter 3, the inverted lists of the OIF can be compressed as the inverted lists of the IF and the metadata table can be used to further reduce space requirements. On the other hand since each original list breaks to smaller ones, we need to keep the first record id of each sub-list explicitly and not as a d-gap from a previous one. As a result, in our implementation, the size the inverted lists in the OIF are only marginally (5%) smaller than the IF lists. Still, the size of the Berkeley DB table for the OIF is significantly larger; the OIF occupies 35% of the space of the original data, whereas the

IF occupies just 22% of the that space. The overhead of the OIF is due to several reasons: the large size of the keys (we did not use prefixes or other compression on them); the fact that we chose to have one B-tree instead of many smaller ones, one for each inverted list; and due to the B-tree fill factor. If a reassignment map is required (i.e., the record ids in the OIF are not the original ids that are used in the relation that stores the data) we would need an extra table taking 8% of the original data, raising the OIF space requirements to almost double than those of the IF (43% vs 22%). Nevertheless, as our evaluation shows, the space overhead of the OIF over the IF does negate its performance benefits.

6.5 Impact of the OIF ordering

An interesting question about the OIF is whether its performance is attributed to the special ordering and the metadata or to the indexing of the inverted lists. To investigate this issue we performed the following experiment. We created a B-tree for the inverted lists exactly in the same way we created the OIF (same block size) but without any ordering for the records. Moreover, we used only the record id as a key for the B-tree instead of the whole records, thus we ended up with a more compact structure compared to the OIF. The resulting unordered B-tree does not have any advantage over the OIF or the IF for superset queries, since the scanning of the whole lists cannot be avoided at each iteration of Algorithm 2. For equality queries (with small selectivities) both the OIF and the unordered B-tree have a similar performance since the candidate solutions are usually very limited and can be directly accessed using the B-tree. The most interesting case is that of the subset queries. We performed an experiment on our default synthetic dataset varying the query selectivity from 10^{-7} (only 1 answer) to 10^{-2}. We created 7 classes of queries with drastically different average selectivities. The first class has all queries that have only 1 answer (10^{-7}) selectivity, the second all queries that have 2-10 answers (10^{-7}-10^{-6}) selectivity and so on up to class 7 with selectivities from 10^{-3}-10^{-2}. In Figure 6.5 we present the average disk page access in each class for the OIF and the unordered B-tree. The results show that the OIF outperforms the unordered B-tree on the inverted lists in all cases. This experiment throws also light to the relation between the OIF and the other structures that offer access to intermediate points in the inverted lists, e.g., skip lists. The ordering and the metadata of the OIF provide a benefit that is orthogonal to the indexing of the lists and other indexing schemes for the lists can benefit for them.

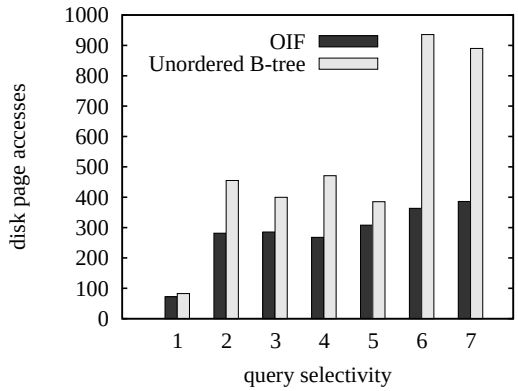

Figure 6.5: Impact of the OIF ordering

6.6 Updates

To present the full picture about the OIF, we evaluated experimentally its performance in the case of updates. We considered only additions to the inverted list (the case of deletions is common for both the IF and the OIF, since no sorting is required). For the inverted file we used the second option described in Chapter 4; i.e., only those lists that must be expanded are affected by the update procedure. We present our results in terms of total evaluation time, since the CPU cost is significant when updating the OIF. In the left plot of Figure 6.6 we depict how does the OIF behave compared to the IF when the number of the updates grows. We fix again our initial dataset to the synthetic dataset that we used in queries, which contains 1M records from a 2K vocabulary that follows a zipfian distribution of 0.8 order. We report the total time for updating the database and the index when inserting in a batch manner collections of 50K, 100K, 200K, and 500K records. All the collections of updates were created with the same characteristics as our basic dataset. The results show that both the IF and the OIF scale in the same manner, since the dominant factor is the total size of the existing index and of the updates. The sorting of the updates happens in main memory and does not have a significant impact on the performance. In the right plot of Figure 6.6 we fix the number of updates to 500K and vary the size of the dataset to 1M, 2M, and 5M, by keeping all other characteristics fixed. Both indices scale linearly but the OIF is significantly more sensitive to the size of the database, since it has to merge it with the updates and re-write it.

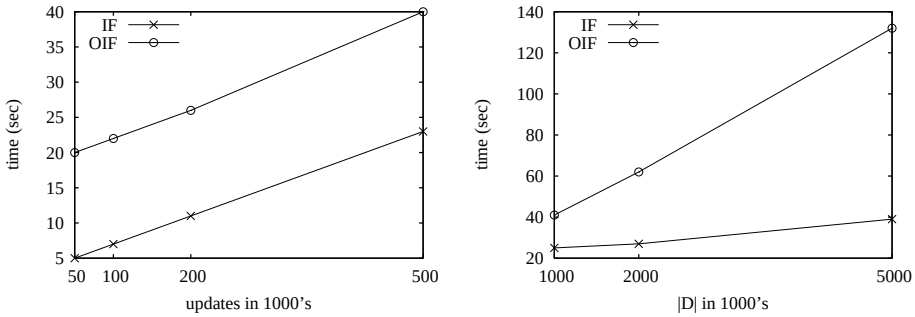

Figure 6.6: Total time for the updates

6.7 Performance summary

It is clear that the OIF offers a different tradeoff between query and update performance compared to the IF. In the example of the dataset with 1M records from a domain of 2000 items, the average query evaluation time (for all three predicates together) took 133 msec for the IF and 25 msec for the OIF. On the other hand inserting 200K records in the same dataset took 12 sec for the IF and 27 sec for the OIF, which results to an average of 0.06 msec per record for the IF and 0.135 msec per record for the OIF. This implies that a workload with a ratio of updates to queries smaller than 766:1, would be handled faster by the OIF. Common applications that use inverted files are not update-intensive, i.e., queries are more frequent than updates, therefore the increased update cost of OIF, compared to IF is gracefully compensated by the great gains in query performance.

In summary, the OIF significantly outperforms the IF in all query cases. The benefits systematically become more clear as the fraction of the database over the domain grows (and the lists of the IF become longer, while the OIF is constrained to few pages due to the appropriate RoI). This is evident both when the database size increases and when the domain is quite small. As a tradeoff to this the OIF is slower than the IF in updates and has increased space overhead, but this disadvantage is leveraged by the fact that in both cases the updates have to be done in a batch manner and also by the small amortized cost of the updates compared to query evaluation.

Chapter 7

Related Work

In this chapter we review the work related to the evaluation of set-containment queries.

7.1 Set-containment queries

Research in set containment query evaluation in a database context has been limited [18, 21]. Recently there has been work on error-tolerant set-containment queries [2]. The error tolerance differentiates the problem significantly from the boolean containment and brings it closer to similarity queries. The authors of [2], differently from our context, focus mostly on main-memory evaluation and not on disk resident indices. The most basic results for containment queries stem from the Information Retrieval (IR) and text databases 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 that have been 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 [7, 28, 31, 36, 37, 43] and similarity queries [17, 29] and less on simple containment evaluation as we do in this paper.

7.2 Inverted files

Inverted files [25, 49] are the indexing solution employed by all WWW search engines and the most efficient index structure for text query evaluation in research literature. In terms of *storage allocation*, the most effective physical storage allocation scheme for inverted lists is to store them contiguously

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

in the disk [47]. Alternatives in the research literature and in practice are motivated by the difficulty to support contiguous storage in the case of updates. To deal with *maintenance operations* 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. All strategies that require that the inverted lists are stored contiguously, store the updates initially in a main memory inverted file. 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 [47]. Several enhancements of the inverted files rely on treating long and short inverted lists differently. Having lists that vary substantially in size, is a result of skewed item distributions in the database, which is a common real world case. The Hybrid Trie Inverted file (HTI) [41], breaks up the larger inverted lists to smaller sublists that contain known combinations of items. This approach provides superior performance over the inverted file for skewed distributions (a comparison with OIF appears in [40]). In [26], König et. al. propose a index structure that similarly to HTI creates inverted lists for combinations of items. It balances the number of indexed item-combinations with the length of the lists, using a cost-model for main-memory access to trade off between these two factors. It is not directly comparable to OIF since it is memory resident and focus only on a specific query (i.e., superset). Finally, in [8] the authors propose an indexing scheme for containment queries that also relies on keeping inverted lists for frequent term combinations, but it is focuses on indexing natural language text documents . Through the suitable choice of term combinations the authors manage to keep the size of the resulting index comparable to the size of a simple inverted index.

7.3 Signature files

Signature files [14, 15] 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. Apart from the traditional signature approaches, the indexing of signatures has also been explored in the literature, with the most prominent idea being the signature trees [13, 29, 9, 33]. Signatures trees organize the records' signatures following the same lines as R-trees and B-trees. Each node has a signature, which is the result of superimposing the signatures of its immediate descendants.

7.4 XML search

Inverted files have also been employed for searching semi-structured data. XML documents can be modeled as trees and their elements and values can be encoded as intervals that capture the ancestor/descendant relationships in the tree hierarchy. Inverted files have been used to index the encodings of XML structural elements in order to efficiently answer XML path and twig queries, which can be modeled as containment joins between the intervals [3, 46]. Follow-up research [24] has shown that indexing the inverted lists can further improve performance. The XML indexing problem for path query evaluation is essentially different to the classic set containment search problems we study in this paper and OIF is not comparable to XML indexes, structurally or operationally.

7.5 Alternative organizations for inverted lists

Special orderings for the postings in inverted lists have been studied in several contexts. In [34] the postings are sorted according to the in-document frequency of each term and in [4, 5, 20] the sorting criterion is the normalized in-document frequency. These approaches focus on the evaluation of IR queries, which require ranking of the results. Ranking queries benefit from these orderings, since most IR similarity measures are directly affected by the in-document frequency of each term. During the evaluation, the algorithm exploits the ordering of the lists to retrieve the results in the desired sort order or to decide on similarity bounds which are used as thresholds for terminating the query evaluation. For similar purposes, inverted lists are sorted according to the record length in [19]. In this case, the record length is used to provide bounds for the similarity between different records. Knowing the bounds allows to limit the search in only a part of the inverted lists. The aforementioned architectures are reminiscent to the OIF, however, their organizations are not suitable for the boolean containment queries studied here [47]. A significant difference also lies in the fact that in these approaches the records are sorted locally in each inverted list, and not globally as in the case of OIF. Finally, in [10], postings are ordered by taking into account spatial information. The main objective is to support combined geographic and keyword search to web documents. Each document can be associated to one or more geographical regions on a map, based on the location of its hosting site, or location information in its content. By encoding these spatial areas in the document ids, querying for documents based on both their textual content and spatial proximity to a location of interest is possible (e.g. find all documents about “yoga” which are associated to locations near Brooklyn). The inverted lists corresponding to the textual query terms are intersected, but regions in these lists that contain document ids not related

to the spatial query component are not accessed. A problem of the previous approaches is that they reduce the effectiveness of compression techniques, since they lead on larger average d-gaps [47].

7.6 List Intersection

Although in this work we focus on the index structure, relevant work to the query evaluation part appears in the literature concerned with list intersection algorithms. In [11, 12] Demaine et.al. describe an adaptive algorithm for computing set intersections, unions and differences. The first algorithm of [12] polls each list in a round robin fashion, and it is ameliorated in [11]. Tsirogiannis et. al. propose in [42] a main memory list intersection algorithm for sorted and unsorted lists. The algorithm exploits the characteristics of modern hardware and focuses on balancing the load between multiple cores. Moreover it probes the lists in order to gather statistics that would allow efficient exploding of the multi-level cache hierarchy. Unlike our approach it does not deal primarily with the disk I/O. Efficient exploitation of multiple core CPU is also in the focus of [39], which proposes inter-query parallelism and intra-query parallelism. The former exploits parallelism between different queries, while the latter parallelizes the processing within a single query. Finally, [6] offers an experimental comparison of several popular methods of list intersection with respect to their CPU cost.

Chapter 8

Conclusions

In this paper we proposed a novel indexing scheme, the ordered inverted file (OIF). The key idea of the index is to order the data suitably to allow early pruning in query evaluation time. We have described query evaluation algorithms that take advantage of the OIF and theoretically showed that the OIF manages fewer disk page accesses and faster evaluation times than the simple inverted file. This claim has also been supported by extensive experiments on both real and synthetic data. The results show that the performance of the OIF is often orders of magnitude superior to that of the inverted file and that it exhibits good scaling properties.

We do not propose the OIF as a total replacement for the inverted file, since in a variety of applications, especially in the IR context, the individual inverted lists are not that big, and the nature of the queries does not permit skipping a significant part of the involved lists. Still, our contribution serves the purpose of answering containment queries with exact semantics, under limited memory budgets. OIF is especially efficient for large collections of records from a limited item domain, or from domains where the item's distribution is skewed. We consider the above assumptions realistic and the OIF to be a practical and superior solution, since it outperforms inverted files for all kinds of containment queries.

Future work can be directed towards the efficient evaluation of composite predicates and other type of predicates (e.g., similarity).

Bibliography

[1] http://www.oracle.com/database/berkeley-db.html.

[2] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, 2010.

[3] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, 2002.

[4] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, 2001.

[5] V. N. Anh and A. Moffat. Impact transformation: effective and efficient web retrieval. In *SIGIR*, 2002.

[6] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:3.7–3.24, 2009.

[7] J.-Y. Cai, V. T. Chakaravarthy, R. Kaushik, and J. F. Naughton. On the complexity of join predicates. In *PODS*, 2001.

[8] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, 2007.

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

[10] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.

[11] E. D. Demaine, L.-O. Alejandro, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, 2001.

[12] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, 2000.

[13] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *SIGIR*, 1986.

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

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

[16] W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[17] A. Gionis, D. Gunopulos, and N. Koudas. Efficient and tunable similar set retrieval. In *SIGMOD*, 2001.

[18] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.

[19] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[20] D. Hawking. Efficiency/effectiveness trade-offs in query processing. *SIGIR Forum*, 32(2):16–22, 1998.

[21] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDBJ*, 12(3):244 – 261, 2003.

[22] S. Hettich and S. D. Bay. The UCI KDD Archive. Univ. of California, Dept. Information and Comp. Science, http://kdd.ics.uci.edu. 1999.

[23] F. M. Institute. Supermarket facts & figures. http://www.fmi.org/facts_figs/superfact.htm, 2006.

[24] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *ICDE*, 2003.

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

[26] A. C. König, K. Church, and M. Markov. A data structure for sponsored search. In *ICDE*, 2009.

[27] N. Lester, J. Zobel, and H. E. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[28] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, 2003.

[29] N. Mamoulis, D. W. Cheung, and W. Lian. Similarity search in sets and categorical data using the signature tree. In *ICDE*, 2003.

[30] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, 2001.

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

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

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

[34] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[35] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[36] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, 2000.

[37] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

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

[39] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *SIGIR*, pages 738–739, 2009.

[40] M. Terrovitis. *Modelling and operational issues in pattern base management systems*. PhD thesis, El. and Comp. Eng., NTUA, available at http://www.dblab.ece.ntua.gr, 2007.

[41] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, 2006.

[42] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.

[43] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[44] H. E. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, 1999.

[45] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

[46] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

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

[48] 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.

[49] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full text databases. In *VLDB*, 1992.