A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes

Manolis Terrovitis
Nat. Technical Univ. Athens
mter@dblab.ece.ntua.gr

Spyros Passas
Nat. Technical Univ. Athens
spas@dblab.ece.ntua.gr

Panos Vassiliadis
Univ. of Ioannina
pvassil@cs.uoi.gr

Timos Sellis
Nat. Technical Univ. Athens
timos@dblab.ece.ntua.gr

ABSTRACT

Set-valued attributes frequently occur in contexts like market-basked analysis and stock market trends. Late research literature has mainly focused on set containment joins and data mining without considering simple queries on set valued attributes. In this paper we address superset, subset and equality queries and we propose a novel indexing scheme for answering them on set-valued attributes. The proposed index superimposes a trie-tree on top of an inverted file that indexes a relation with set-valued data. We show that we can efficiently answer the aforementioned queries by indexing only a subset of the most frequent of the items that occur in the indexed relation. Finally, we show through extensive experiments that our approach outperforms the state of the art mechanisms and scales gracefully as database size grows.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*Access Methods*

General Terms

Algorithms, Performance

Keywords

HTI, inverted files, tries, containment queries

1. INTRODUCTION

Containment queries on set-values emerge in a variety of application areas ranging from scientific databases to XML documents. Examples of set valued data can be found in market basket analysis, production models, image and molecular databases [7]. Containment queries span a wide range of query families, ranging from simple existence queries to composite similarity, pattern matching, or graph isomorphism queries. Naturally, the fundamental set-containment operators are typical for a large number of situations (e.g., *"Give me all photographs whose annotation contains the terms 'galaxy' and 'red giant'"*, or *"Give me all protein sequences that contain either 'G' or 'T' or a combination of them, but nothing else"*). Moreover, set-containment operators can be used in other query classes where a pruning of the candidate sets to be processed takes place (e.g., *"Give me all medicines sequences that are similar to my XYZ test medicine and their X component contains either 'G' or 'T' or a combination of them, but nothing else"*). Another important application area for containment queries is the evaluation of path expressions in XML data, which partially resolves to keyword searching [9]. As RDBMSs and IR come closer, often in the interest of storing and handling XML [20] and web data [4], containment queries on set values become a more and more significant use case for an RDBMS.

A natural way of modelling and storing set-values in modern RDBMS is by using set valued attributes. Set valued attributes are an integral part of the object-relational model and they are supported by most modern RDBMS's [17]. In this context, we are interested in containment queries over the set valued attributes of a relation. More specifically, assuming a relation $D(id, set\_values)$ and a set of interesting items $qs = \{i_1, \ldots, i_n\}$, we would be interested to ask queries of the form $\{t \mid t \in D \wedge qs\ \theta\ t.set\_values\}$, where $\theta \in \{\subseteq, \equiv, \supseteq\}$. The problem of efficiently computing the result set of these operations is challenging, mainly due to the vastness of the underlying data volumes and the particularities of the queries. The problem with set values is that the space of potentially indexed values is enormous (2^n, for n items) and the resulting index would also be huge as well. Moreover, the query semantics are quite different: whereas simple subset queries retrieve the tuples that contain a certain set of items, superset values require that some (but not necessarily all) of these items are contained in the result tuples, and nothing else. Therefore, an efficient indexing scheme that can (a) support the coexistence of multiple items in the same query set and (b) adequately support different classes of containment queries by exploiting their characteristics is necessary.

To this day, the database and the information retrieval(IR) research communities are mainly the ones having studied set-values in depth. From the database perspective, there is a need to efficiently handle huge volumes of small sets, usually taking values from a limited domain. So far, database research has mostly focused on similarity and join queries. Similarity queries [3, 12] retrieve the set values that are most similar to the one provided in the query. Join queries, which are classified as similarity joins [15], or as set containment joins [11, 13], focus on intersecting two different relations based on their set valued attributes. Research on access methods for basic containment queries is very limited. To

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.
Copyright 2006 ACM 1-59593-433-2/06/0011 ...$5.00.

the best of our knowledge, only access methods based on signature files [2] and inverted file indices [10, 22] have been used in the database research literature for supporting containment queries on set valued attributes.

A recent survey [7] has shown that inverted files clearly outperform signature-based methods for containment queries on low cardinality set values. The same holds for text documents as Zobel et. al. showed in [21]. Moreover, Zhang et. al., studied in [20] how inverted files indices compare to traditional relational methods for containment queries, motivated by the integration of IR functionality in RDBMSs. Using traditional relational indices like B-trees for containment queries was shown to have significantly inferior performance in most cases. Considering inverted files as the state-of-the-art mechanism for set containment is also supported by the fact that they used by all WWW search engines [19].

Still, the performance of inverted files suffers when the domain of the distinct items of the database is small or when the distribution of the items is skewed and few items dominate the dataset. 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 transactions that contain this item is maintained. Thus, if some items appear in many set values, their inverted lists become very long. Since containment queries usually require scanning the entire inverted lists of the query items, having long inverted lists has a deteriorating impact of the query evaluation. This is often the case of real world. A characteristic case of numerous records of set values from a limited domain are the real datasets from UCI KDD archive [8] that we use in our experimental evaluation. 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?"*). Moreover, highly skewed data is a common case for retail transactions, where some basic products dominate the transactional logs.

In this paper, we focus on the efficient evaluation of containment queries on large collections of low cardinality sets with exact query semantics. The query classes under investigation include *subset*, *superset* and *set equality* queries. These queries test a set of items, *a.k.a query set*, over a set valued attribute of a set of records, for the fulfillment of the query's selection condition (subset, superset or equality). The exact set of transactions that fulfill the selection condition is returned. To efficiently answer these classes of queries, we propose a novel indexing scheme, the Hybrid Trie-Inverted file (HTI) index. The HTI-index superimposes a trie structure, the *access tree*, over an inverted file index. The access tree offers pointers to the inverted lists of the most frequent items, thus leveraging the performance of inverted files. In the HTI index, queries over the frequent items are evaluated by the access tree. 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. This evaluation mechanism has a significant impact on query answering efficiency in the average case, since we expect items to be 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 transactions 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 quickly identify the set of frequent items that participate in the query 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 transactions, 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.

The rest of the paper is organized as follows: In Section 2 we formulate the problem and in Section 3 we present the proposed HTI index. Section 4 describes the query evaluation algorithms and in Section 5 we demonstrate the results of the experimental comparison of our proposal against the inverted file index. Finally, Section 6 concludes the paper.

2. PROBLEM FORMULATION

For reasons of simplicity we assume that the data are organized under a simple object relational schema D with each tuple $t = [id, s]$ having two attributes; id is a unique identifier of the transaction and s is a set (not a bag or a list) of objects from an infinitely countable domain of distinct items. We refer to the active domain of D, with the term *vocabulary* and denote it as I. Thus, every $t.s \subseteq I$. Moreover, throughout the paper we consider the id as adequate information to allow us to retrieve the whole transaction from the hard disk in one step (i.e., in one page access).

Queries. In queries on set valued data, the user specifies the query predicate and the *query set* qs. The query set is a set of items from the domain of I. The queries we are interested in are defined as follows:

- *Subset queries.* In subset queries the user asks for all transactions 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 transactions that *contain exactly* the query set, i.e., $\{t \mid t \in D \wedge qs \equiv t.s\}$.

ID	Items bought
1	{f, a, c}
2	{c, b, d}
3	{f, a}
4	{a, c}

ID	Items bought
5	{f, d}
6	{f, c}
7	{f}

Figure 1: Example relation D of customer transactions

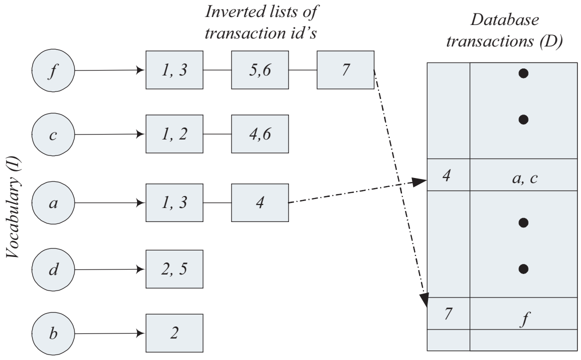


Figure 2: A simple inverted file index scheme for the example of Figure 1.

- *Superset queries.* In superset queries the user asks for all transactions whose items *are contained* in the query set, i.e., $\{t \mid t \in D \wedge qs \supseteq t.s\}$.

3. INDEX STRUCTURE

Tries and inverted files have been extensively used for text indexing, still 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 give background information for inverted files and tries and we explain their benefits and drawbacks. Then, we show how these indexing schemes are combined in the HTI index. Finally, we also discuss issues concerning updates, compression and caching.

3.1 The inverted file

The 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 2. Each list node has a label indicating the item it represents and a pointer to the head of the inverted list. The inverted list contains information about all the transactions in which the item appears. In our case, this information comprises the transaction *id* alongside with its length. The length of the transaction is required in order to efficiently execute equality and superset queries.

Figure 2 depicts an inverted file index for the relation of Figure 1. The vocabulary includes all the distinct items that appear in the transactions. The inverted lists may be huge for large databases; 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 transaction collection or even larger. Uncompressed inverted files for text documents typically consume around 30% of the space required for the uncompressed database [16]. In the cases that we are mostly interested, where there are no repetitions and the vocabulary is significantly smaller than the number of transactions ($I \ll D$), the inverted file can be equal or larger than the database, since the $t.id$ requires more bits than the items of I.

We can trace the answer to subset, superset and equality queries by using set operations on the inverted lists. Due to their size, the inverted lists are stored in secondary storage. Therefore, the larger these inverted lists are, the more memory pages have to be retrieved from the disk for evaluating a query. This means that the most frequent items that have the larger inverted lists are the most expensive to process. This is an important weakness, when dealing with set values in databases, considering that most frequent items are usually the ones most frequently queried.

3.2 Tries in the context of set-values

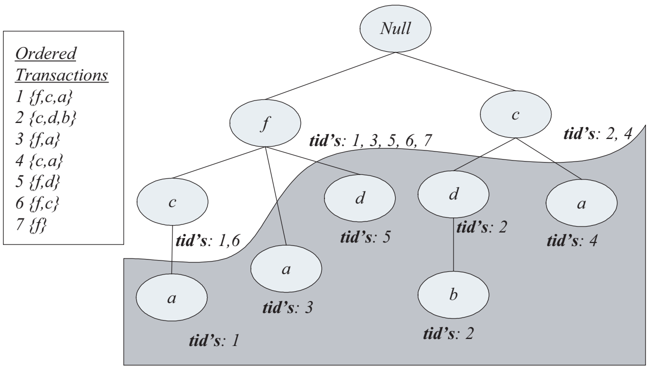
Tries are multiway tree structures for storing string keys which enable retrieval in time proportional to the string length [1]. Unlike inverted files, tries are letter oriented and each string corresponds to a path in the tree. Consequently, common prefixes in strings correspond to common prefix paths 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 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 the space gain achieved from common prefixes is a lot smaller compared to the one in the text document case. Practically, even for a moderately large I, e.g., 20k, the maximum space of the trie is so big, that it grows almost linearly with the number of transactions.

In our following deliberations, we need to define the fundamental notion of item frequency ordering that concerns the ordering of the items of a vocabulary.

Item frequency ordering. The *item frequency ordering* of the items of a vocabulary I (over a database D) is the total ordering of the items according to their frequency of appearance in the underlying database. In our reference example, the item reference ordering $<_I = [f, c, a, d, b]$.

To construct a trie for set values, we follow the approach of Han et al. in [5, 6]. First, each transaction is transformed from an unordered set to an ordered *sequence* based on the item frequency ordering of the vocabulary. An item x precedes another item y in an ordered transaction if x is more frequent than y in the whole database D. The ordered transaction 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 transaction, we only



How the full trie would ideally be. The shaded area is to be excluded in the access tree of HTI

Figure 3: An abstract form of a trie tree for the example of Figure 1

add the new nodes.

An abstract form of the trie tree for the database of Figure 1 is depicted in Figure 3. The transaction with $id = 1$ and set value $s = \{a, f, c\}$ is ordered according to the frequency of its items in the database. Since f occurs 5 times, c occurs 4 and a occurs 3 times, the transaction's set is transformed to a sequence $s = \{f, c, a\}$ that subsequently contributes the path $f \rightarrow c \rightarrow a$ in the trie. Unlike typical tries, in Figure 3 we annotate each node with the list of transaction id's that correspond to it (without implying that they are actually kept in main memory along with the trie). Note that depending on its prefix, a transaction might belong to the list of more than one nodes. For example, the transaction with $id = 1$ belongs to the lists of all the nodes of its prefix, i.e., all the nodes of the path $f \rightarrow c \rightarrow a$. Finally, there is a difference among the transactions that pertain "solely" to a node and the transactions that also pertain to its descendants. Observe the node c of the path $f \rightarrow c \rightarrow a$. The transaction with $id = 1$ is the transaction $\{f, c, a\}$ that also belongs to the node a of the same path. On the contrary, the transaction with $id = 6$ refers exactly to the path $f \rightarrow c$. 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, does 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.

3.3 The HTI index

As we have explained in Section 3.1, 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 4.

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 transactions that share the same *access prefix paths* (*app*). The *app* of a transaction 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 transaction. For example, the *app* of $\{f, a\}$ is $\{f\}$. We store the *app* of each transaction in the access tree, by putting the first and most frequent element as a direct child of the root (see 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 transactions that contribute to the path from the root to the node. These are all the transactions 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 4 we depict an example HTI index for the relation of Figure 1. We choose as frequent items $I_{fr} = f, c$ (having a frequency greater than 3), and we create the access tree considering only them. Observe that in Figure 3, these were also the items with the longest transaction lists. The shaded area in Figure 3 concerns the infrequent items that were subsequently dropped from the access tree of Figure 4. Item f is more frequent than c, thus it precedes it in access tree paths. Assuming this I_{fr} set, all the transactions of Figure 1, contribute to three paths: *root* $\rightarrow$ f,

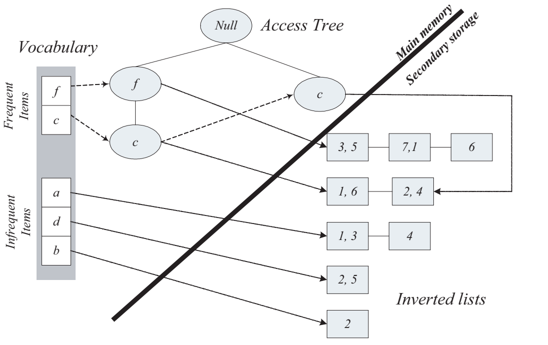
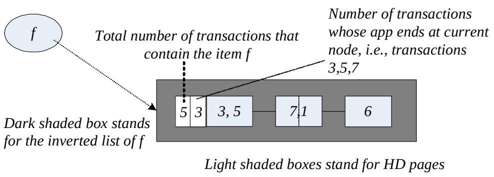


Figure 4: HTI index for the relation of Figure 1.



Total number of transactions that contain the item f

Number of transactions whose app ends at current node, i.e., transactions 3,5,7

Dark shaded box stands for the inverted list of f

Light shaded boxes stand for HD pages

Figure 5: The transaction list corresponding to the node f, assuming two id's per disk page.

$root \rightarrow f \rightarrow c$ and $root \rightarrow c$. Observe, also, how the nodes labeled c 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 transactions 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, for the case of f: (a) the $id's$ of the transactions whose *app* ends at this node; these are transactions $id's$ 3,5 and 7, (b) the $id's$ of rest of the transactions that contribute to the current node; these are $id's$ 1 and 6. In the beginning of each information node sublist, we store the number of transactions of case (a) alongside with the total number of the transactions that contribute to the current node, so that we can retrieve the right block from the disk each time.

Example. As shown in Figure 4, the access tree and the vocabulary are kept in the main memory, whereas the inverted lists reside at secondary storage. Transactions 1,3,5,6,7 contribute to the path $root \rightarrow f$, thus they are stored at the inverted sublist of node f. Observe that f, 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 c has two sublists. Two of the transactions of f, 1 and 6, also contribute to the path $root \rightarrow f \rightarrow c$, and they are stored in the first inverted sublist. At the same time, transactions 2 and 4 contribute to the path $root \rightarrow c$ and they are stored at the second sublist. For storage efficiency, the individual sublists of all the different nodes of c 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 4 we depict only the id's that are contained in the sublists and not the labels that mark each sublist for reasons of readability. The real structure of the sublist for the case of item f is depicted in Figure 5.

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

Updates in HTI index. When a new transaction is to be inserted or deleted from the HTI index we practically have to perform two different updates: one to the inverted file component and one to the access tree. If the items and the order of I_{fr} are not modified, the case is straightforward [18]. Still, it is also possible that the order and the member items of I_{fr} should be changed, due to changes in the items appearance frequency. In all such cases, the query evaluation algorithms are correct. Considering also, that in most related application areas, the relative frequencies of the items change slowly or remain stable, the rebuilding of the index is not necessary. In any case, the frequency ordering reflects a heuristic for keeping the size of the access tree small, as reported in [5]; other orderings could also apply. For more details on the creation and maintenance of the HTI index we refer the interest reader to the long version of the paper [18].

Compression and caching There is a question of how the HTI index compares to inverted files, when compression techniques are applied [16, 14] 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. 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 transactions do not affect its size and effectiveness. Thus, for small vocabularies and especially for skewed distributions, the HTI index is a better choice.

4. 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.

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.

4.1 Subset queries

Subset queries are the most common queries executed against transaction 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 transactions 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 transaction 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 transaction *id*'s that correspond to the infrequent items $i_{k+1}, \ldots, i_n$ with the list of the already retrieved $candidateIds$. Any transaction *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 6.

```
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 transactions 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 transactions 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 6: Algorithm for determining subset queries

Assume for example that the user asks for all transactions that contain the $\{f, c, a\}$ items from the relation D depicted in Figure 1. If we evaluate the query against the inverted file, depicted in Figure 2, 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 $t.id = 1$. If, instead, we evaluate the query against the HTI index, the disk pages accesses are much less. First, we have to identify the *app* of the *qs* which is $f \rightarrow c$. 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: $root \rightarrow f \rightarrow c$. Now we can directly retrieve the transactions that contain f and c, which are 1 and 6 by performing only 1 page access. Subsequently we can merge-join $\{1, 6\}$ 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 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 algorithm of Figure 6. 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.

4.2 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 transactions are filtered according to their length, which must be equal to $|qs|$. We refer the interested user to the long version of the paper [18] for the pseudocode of the evaluation algorithm. 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.

4.3 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 $l_1, \dots, l_n$ disk pages respectively, evaluating a superset query solely in the inverted file, with the algorithm presented in Figure 7, 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 transactions that *end* in any node of these paths. For these transactions we know that they do not contain any other item of I_{fr}, except from $f_1, \dots, f_k$. If the qs has non frequent items too, then we have to check in the inverted file if the remaining items of the transactions of *candidateIds* contain only items from $i_{k+1}, \dots, i_n$. 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 transaction itself, the transaction 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 7.

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 transactions whose *app* ends at the access tree nodes, as opposed to the rest of the transactions within the same inverted sublist.

5. EXPERIMENTAL STUDY

As several surveys and previous research have demonstrated, the inverted files, although a simple technique, offer better performance than signature based methods for low cardinality set values [7] and for document indexing [21]. Moreover they outperform traditional indices like B-trees, for containment queries in RDBMSs [20]. For the aforementioned reasons, we chose the inverted files as the main point of reference for the evaluation of the HTI index.

5.1 Methodology

HTI index. We have implemented a prototype of the HTI index according to the description we gave in Section 3. Since query evaluation performance is dominated by disk accesses, our implementation is aimed at providing accurate results on number of disk pages accesses during query evaluation on the HT-index.

Some aspects of the index functionality were simulated; disk pages are $4k$ arrays in main memory, and sibling nodes are stored in linked lists instead of arrays. This implementation provides accurate results both on the page accesses and on the size of access tree in the main memory. The former are explicitly counted by the program and the latter can be computed by ignoring the links between sibling nodes.

```
Algorithm SupersetQueries
Input: An $HTI$ index $H$ over a dataset $D$, a query set $qs = \{f_1, \dots, f_k, i_{k+1}, \dots, i_n\}$ and a query $Q = \{t \mid qs \supseteq t.s\}$.
Output: the $t.id$'s of the transactions that where $t.s \subset qs$
Method:
  Determine the $app = \{f_1, \dots, f_k\}$ of the query set.
  If $app$ is not empty use supersetTrie($app$,$root$) to retrieve the
                                        $candidateIds$ from the trie.
  Let $il_1 \dots il_m$ be the inverted lists of all the non frequent
    items of the $qs$ and the $candidateIds$, ordered according to
    the number of memory pages

  for ($i$=1 ; $i \leq n$ ; $i$++)
      for each entry $t$ of $il_i$
          $unmatched$=$t.length - 1$
          if ($unmatched$ == 0) add $t$ to $result$ and break
          for ($j = i + 1$ ; $j \leq n$ ; $j$++)
              if ($unmatched > n - j$) break
              if ($unmatched$==0) add $t$ to $result$ and break
              scan forward $il_j$
              if $t$ found in $il_j$ $unmatched = unmatched - 1$
  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 transactions
whose items are contained in $app$
Method:
  while ($app$ not empty)
      $newCNode$=pop($app$)
      if $newCNode$ is child of $currentNode$
          add the inverted sublist of $newCNode$ to
                                        $candidateIDs$
          supersetTrie($app$,$newCNode$)
  return $candidateIds$
```

Figure 7: Algorithm for determining superset queries

Inverted files. We have implemented a basic version of the inverted file index. The vocabulary is kept in a hash table and the inverted lists in $4k$ arrays corresponding to disk pages. Each entry in the inverted file comprises the *id* and the *length* of each transaction. The size of each entry is $e_s = sizeof(long\ int) + sizeof(short\ int)$, which is 6 bytes in our case.

Real data. We have evaluated HTI on two real datasets from UCI KDD [8] archive. Both of them are logs of user behavior on web portals. The first one, denoted as *msweb*, is a one-week log tracing the virtual areas that users visited in the web portal `www.microsoft.com`. Each record corresponds to a user session and the set value comprises the areas she/he visited. 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 items. Since the dataset is small, to illustrate the performance of the two indices better, we created a new one, by duplicating the records by a factor of 10, which resulted to a dataset of 320k records. This multiplication is reasonable, since it simply corresponds to a 10 week log.

The second dataset, denoted *msnbc* is again a log of users behavior on the web portal of `msnbc.com` taken from the UCI KDD archive as well. 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 items.

Synthetic data. To investigate how HTI behaves for datasets and domains larger than the ones we had from real sources, we used synthetic data, with a skewed zipfian distribution of order 1 (as in [7]). Duplicates in each transaction were dropped and we ended up with transactions with lengths from 2 to 22 items, uniformly distributed.

Query generation. We created query sets for all the three types of queries. As in other approaches [7], we consider the evaluation of the proposed method on queries that always have a solution as more informative. We created such queries by randomly selecting existing transactions from D.

For the synthetic data, we ranged the number of items in the query set, $|qs|$, from 2 to 22 and we created 50 queries of each type. For the real data, we ranged the $|qs|$ from 2-7, since their domain and the average record length is a lot smaller. The selectivities of the subset queries are less than 3%, with highest appearing for queries 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.

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 dominating 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.

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.

5.2 Performance of the HTI index

5.2.1 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 8 and 9. 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 a order of magnitude better for all types of queries.

5.2.2 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 10 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 10, 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 $|I|$. Individual results obey the general trend and are omitted for the interest of space.

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 11 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.

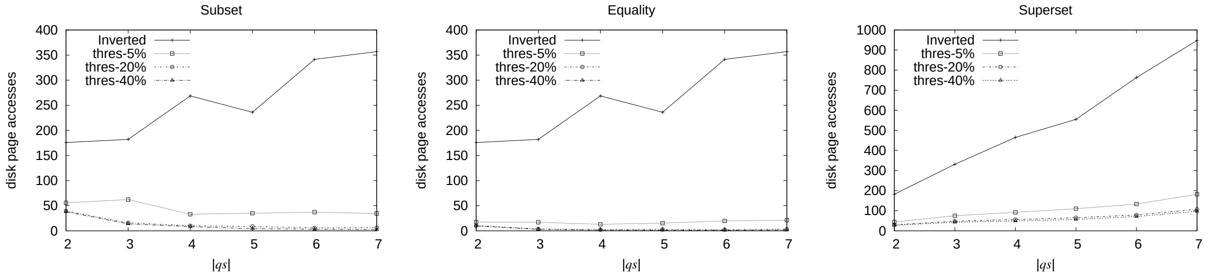


Figure 8: Average performance of queries on *msweb* data

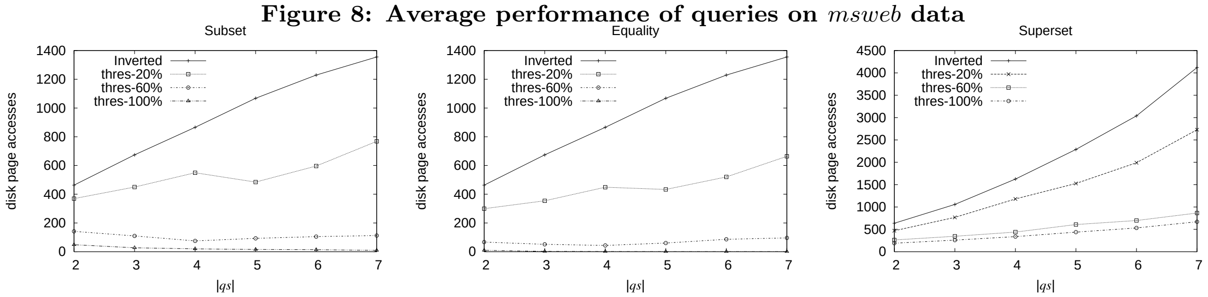


Figure 9: Average performance of queries on *msnbc* data

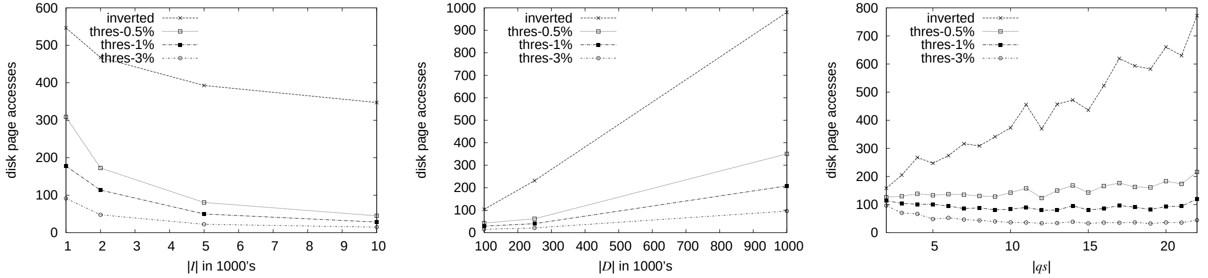


Figure 10: Average performance of subset queries

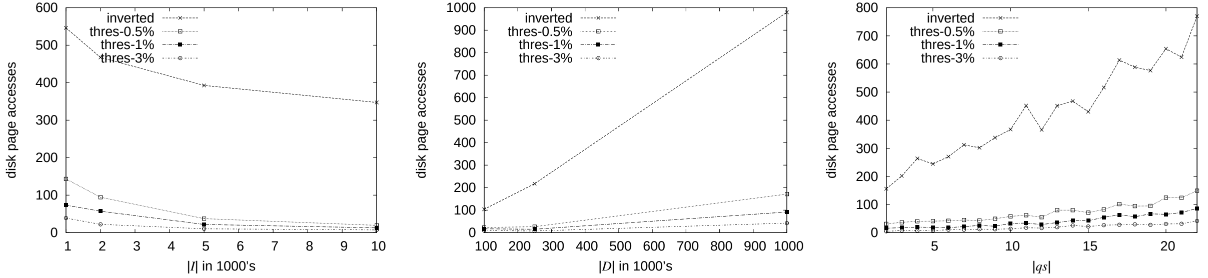


Figure 11: Average performance of equality queries

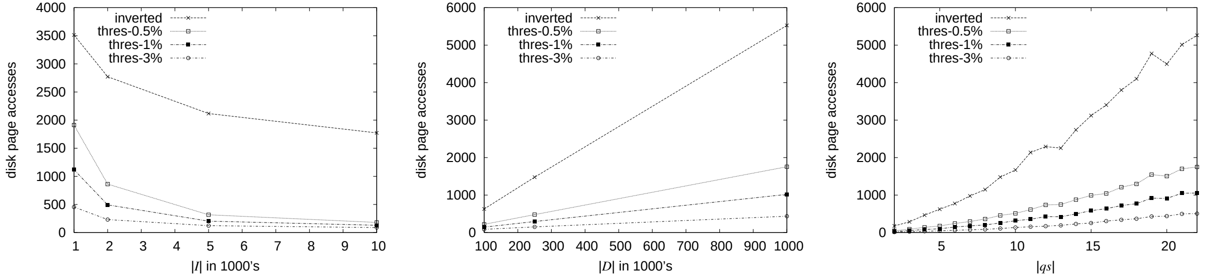


Figure 12: Average performance of superset queries

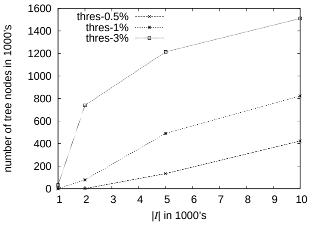


Figure 13: Effect of $|I|$

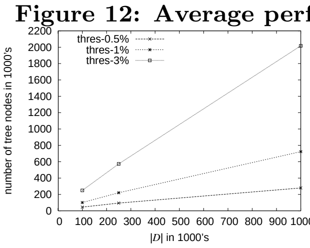


Figure 14: Effect of $|D|$

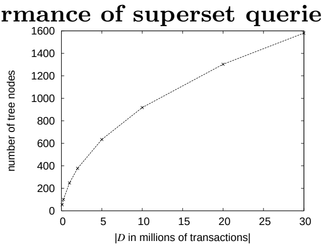


Figure 15: $|I| = 5k$, 0.5%

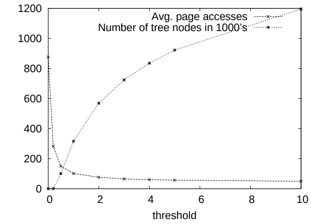


Figure 16: Effect of k

Superset. As it can be inferred from Figure 12 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.

5.3 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 13 and 14 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 15, 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.

5.4 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 16 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.

6. CONCLUSIONS

In this paper we have tackled the problem of containment queries on large collections of low cardinality set-valued attributes. We have proposed a novel indexing scheme, the HTI index, which superimposes a trie tree (kept in main memory) over an inverted file (kept in secondary storage) to efficiently answer subset, superset and set-equality queries. We have introduced novel evaluation algorithms for these classes of queries that use the HTI index and experimentally demonstrated that the HTI clearly outperforms the state-of-the-art organization scheme, i.e., the inverted file, with reasonable main-memory overhead. Our experiments have showed that the scale of our approach is a lot smoother than the one of inverted files and in certain cases, for large database or query-set sizes, we can reduce the disk page accesses by orders of magnitude, with a small overhead of main memory.

Future work comprises further investigations on how to reduce the size of the access tree and how to exploit the HTI index to efficiently support other kind of queries, like, for example, set intersections or similarity queries.

7. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

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

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

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

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

[6] 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.

[7] 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.

[8] S. Hettich and S. D. Bay. The UCI KDD Archive. University of California, Department of Information and Computer Science. 1999.

[9] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2004.

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

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

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

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

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

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

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

[17] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1996.

[18] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. HTI technical report. www.dblab.ece.ntua.gr/~mter/papers/TR-HTI-01.pdf, 2006.

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

[20] 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.

[21] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM TODS*, 23(4):453–490, 1998.

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