Indexing Heterogeneous Data

Nilesh Dalvi
Department of Computer Science
University of Washington
Seattle, WA
nilesh@cs.washington.edu

Dan Suciu
Department of Computer Science
University of Washington
Seattle, WA
suciu@cs.washington.edu

September 27, 2003

Abstract

We consider the indexing problem for heterogeneous data, where objects are sets of attribute-value pairs, and the queries specify values for an arbitrary subset of the attributes. This problem occurs in a variety of applications, such as searching individual databases, searching entire collections of heterogeneous data sources, locating sources in distributed systems, and indexing large XML documents. To date no efficient data structure is known for such queries. In its most simplified form the problem we address becomes the *partial match* problem, which has been studied extensively and is know to be computationally hard. We describe here the first practical technique for building such an index. Our basic idea is to precumpute certain queries and store their results. User queries are then answered by retrieving the "closest" stored query and removing from its answers all false positives. The crux of the technique consists in chosing which queries to precompute. There are several design choices, and we evaluate them both theoretically and experimentally (with real data sets). One major advantage of this technique is that it can be easily implemented on a relational database.

1 Introduction

Web search engines have proved highly popular for searching unstructured documents (Web pages). While in principle a similar technology can be deployed to search structured information sources, this has not happened so far. We believe that there are two reasons for that. First, there was little incentive. Structured information sources are either beyond firewalls (e.g. corporate data bases), or unique, well known data sources that are queried by users directly, without any support form a centralized index (e.g. the Protein Sequence Database, `http://pir.georgetown.edu/`). The second reason is technical: it is much harder to search and index a large collection of structured objects than a collection of documents.

We argue that the first reason is quickly disappearing, as more communities need to share their structured sources on the Web. Consider the case of sharing scientific data. Until recently, individual researchers collected and analyzed their own data and kept it proprietary, except for a few large, well-known and well-maintained public databases. But today there is increased pressure on the scientists to share their data. The U.S. National Institute of Health will expect all funded scientists to release their final research data for use by other scientists [23]. Some scientific journals are making data sharing a condition of publication [22], while others are coming under pressure to do so [19]. Examples of efforts by scientific communities to share structured information sources are the SkyQuery project in astronomy [26, 20], the Institute for Systems Biology [15], and the Human Brain Project. In the business world, vendors of commercial database systems are also considering adding keyword searches to their systems. While most commercial systems already provide full text capabilities, vendors are now moving towards supporting searches in both the schema and the content of a database: DataSpot [9] is a middleware, DBXplorer [3] is an extension of Microsoft's SQL Server, while DISCOVER [14] describes a general framework for keyword searches in a database and demonstrates it on Oracle. These systems offer keyword searches over a single database. The next, and harder step, is to support keyword searches on collections of data sources: this is needed for example in order to locate relevant sources in large distributed systems [11].

The technical barrier however remains: searching structured data sources is a harder problem than searching unstructured documents. One challenge is that the translation of keyword searches into SQL becomes too difficult: the techniques used by the systems above for a single relational database do not scale to a large collection of data sources. Another challenge is how to reconcile and relate the schemas of the various data sources. There has been extensive work in the past on addressing schema mappings [21, 13], and we will not be concerned with that in this paper. Even assuming the schemas have been reconciled, it is unclear how one would search the collection of heterogeneous data. Unlike information retrieval systems, here we need to support queries that specify schema elements in addition to data values being searched. In other words the queries need to be schema aware.

Our approach to searching heterogeneous data sources consists of two steps. In the first, all data sources are mapped into a common data model, in which objects are represented as sets of attribute-value pairs. While this translation has the disadvantage that all data needs to be copied to a central server, and may not be fully up-to-date with the original data sources, it has the advantage that sources can be easily added and schema mappings can be easily encoded in the translation itself. More importantly, it addresses the query translation problem: keyword searches become now *partial match* queries. A partial-match query consists of a set of attribute-value pairs, and its answer consists of all objects that contain all these attribute-value pairs, and possibly more: hence the name "partial-match". The second part of our approach deals with techniques for efficiently answering partial match queries. This will be the focus of our paper. The specific technical problem that we address is how to build an efficient index for partial match queries on heterogeneous data. Partial match queries have been extensively studied in the past, starting with Bentley [4], mostly in the context of relational data, and have been considered to be difficult to answer efficiently. In the case of heterogeneous data the problem becomes even harder, because the total number of attributes is much larger, and recently some lower bounds on its complexity have been proved [6, 16].

We describe in this paper the first *practical* index structure for partial match queries on heterogeneous data, called PMI, for Partial Match Index. In our approach, the answers to a certain subset of queries are precomputed and stored in the index. The selected queries need to be chosen carefully, in order to allow *all* queries to be answered efficiently using the stored answers, and at the same time preventing the size of the index from growing out of control. An important point for the class of applications that we envision, is that the partial match index can be implemented easily on a relational database (we used a popular commercial database engine), and a partial match query can be answered with just a few disk accesses. The paper makes the following contributions:

- It describes an approach to searching heterogeneous data through partial match queries.
- It proposes a simple index structure for partial match queries, that can index an arbitrary set of heterogeneous data.
- It describes how this index can be implemented using a relational database engine.
- It analyzes the time complexity of the index, and discusses several variations to the basic scheme.
- It evaluates the index structure experimentally, on real data sets.

The paper is organized as follows. Section 2 defines the problem and shows the limitations of naive approaches. The index is described in Section 3. It is then analyzed and discussed in Section 4. Section 5 presents the experimental evaluation. We discuss related work in Section 6 and conclude in Section 7.

2 Searching Heterogeneous Data

We describe here our approach to searching heterogeneous data.

2.1 Problem Definition

Data Model The first step is to map the heterogeneous data into a flat semistructured data model where objects are sets of attribute-value pairs. Thus, a data object d is:

$$d \quad ::= \quad [A_1 = v_1, A_2 = v_2, \ldots, A_k = v_k] \tag{1}$$

where $A_1, \ldots, A_k$ are *attributes* and $v_1, \ldots, v_k$ are *atomic values*. The set of attributes is not fixed a priori: an attribute is just a string. Each object may have a different set of attributes, and multiple occurrences of the same attribute are allowed.

Data sets can typically be mapped to this simple model quite easily, e.g. by outer-joining relations in a relational model, or flattening elements in an XML instance. For a single relational database the attribute-value pair model corresponds to the *universal relation*, described for example in [27]. In order to be able to map multiple data sources to this model it is important to keep the model flexible and to allow each object to have its own set of attributes. When schema mappings between sources are available, then attributes have to be renamed, to ensure that attributes from different sources that are mapped to each other have the same name in the attribute-value pairs model. Notice that the model is flat, and, as a consequence, some information may be lost during the mapping, e.g. grouping associations in nested XML documents.

Each object in (1) has a unique object-identifier, oid. We denote with U a set of such objects: U will be our data instance.

A *partial-match query* has the form:

$$q \quad = \quad [B_1 = w_1 \ \texttt{ and } \ \ldots \ \texttt{ and } \ B_p = w_p] \tag{2}$$

Here $B_1, \ldots, B_p$ are attributes and $w_1, \ldots, w_p$ are constants. Each condition $B_i = w_i$ is called a *predicate*, hence the query consists of a conjunction of p predicates. The query q *matches* a data object $d \in U$ if all attributes in q occur in d, and have the same values: formally, referring to Equations (2) and (1), if there exists integers $i_1, \ldots, i_p \in \{1, 2, \ldots, k\}$ s.t. $B_1 = A_{i_1}, \ldots, B_p = A_{i_p}$ and $w_1 = v_{i_1}, \ldots, w_p = v_{i_p}$. The query specifies only parts of the attributes and values in the object, hence the name partial-match query. For a query q and data instance U, we denote with $SUP_U(q)$ the set of objects in U that match the query q, and refer to it as the *support* of q.

Applications The partial match problem has numerous applications. We illustrate here three. First, consider the special case when there is a single attribute, call it A. Then both objects and queries are sets of strings. A query q matches an object d iff all strings in q appear in d. This is precisely the setting in information retrieval engines, where both documents and queries are sets of words. Hence any efficient implementation of partial match queries also applies to IR engines.

The second application is indexing XML documents. Previous work [8] focused on the navigation part of XPath queries, by indexing path queries with a single predicate. To evaluate XPath expressions with multiple predicates, the results of each individual predicate must be joined. A technique that replaces joins with traversal of a suffix tree is described in [28]. Partial match queries offer an alternative approach to XML indexing, in which the predicates are evaluated first, then the navigation constraints are checked separately. To evaluate the predicates, an XML document is first mapped to the attribute-value pairs model by associating an attribute A_i, $i = 1, \ldots, k$ to each path occurring in the document. A partial match query specifies a conjunction of predicates on the XML documents being searched.

The third application is peer data management systems [12, 13, 17, 1], which propose a flexible architecture for sharing heterogeneous data. A major unsolved problem in such systems is how to discover peers that hold relevant data. Most systems proposed so far use some form of flooding or gossiping, which has known limitations. An exception is [11], which uses a distributed hash table (DHT) to locate data sources in a distributed system. None of these systems scale well to queries with multiple keywords, which are precisely the partial match queries.

The Problem This paper addresses the following:

Problem 1. *(Partial Match Indexing Problem) Given a data instance U, construct a data structure (the index) s.t. for every partial match query q the answer $SUP_U(q) = \{o_1, o_2, \ldots, o_k\}$ can be computed efficiently.*

Example 2.1 We illustrate the partial match problem with an example from indexing XML data. Consider the Protein Sequence Database, available at `http://pir.georgetown.edu/` which is an XML data instance containing entries about proteins. We map this XML data to our data model by flattening it[1]. Every sequence of XML tags becomes a new attribute, for a total of 75 attributes, like:

```
header/created_date
reference/refinfo/authors/author
reference/refinfo/year
organism/variety
. . .
```

The data is truly semistructured: different protein entries have different sets of attributes, and some attributes may have multiple occurrences in the same entry. Consider now a typical partial match query:

```
[refinfo/year = ’2000’ and
 reference/refinfo/authors/author = ’White,_O.’
 and reference/refinfo/citation = ’Nature’
 and feature/feature-type = ’domain’]
```

Such queries are often formulated by scientists when searching scientific databases. They use multiple predicates to narrow their search, much in the same way we use multiple keywords in Web search engines.

The data set has about 260,000 objects (protein entries); the query matches only 43 objects. The challenge in designing a partial match index is to be able to retrieve these 43 objects with a reasonably small number of disk I/Os.

What is interesting about this query is to examine the number of objects that match each individual predicate. This is shown in the table below (we show only the last element label, for succinctness):

Predicate	No. of Matches
`year=’2000’`	37,335
`author=’White,_O’`	37,822
`citation=’Nature’`	63,220
`feature-type=’domain’`	35,004
all four predicates	43

Each individual predicate matches more objects (by three orders of magnitude) than the entire query. As we show next, this gap makes the partial match indexing problem hard.

[1]Some information about grouping is lost during flattening.

2.2 Naive Approaches and Their Limitations

We discuss here a few simple approaches to the partial match indexing problem and highlight their limitations. We start by considering approaches based on a relational database engine, then briefly discuss other techniques. Recall that U denotes the data instance.

The Join Approach Define the *data table* to be a relation `RU(Oid, Attr, Val)` which stores all attribute value pairs for all objects in U. That is, an object $d = [A_1 = v_1, \ldots, A_k = v_k]$, having identifier o, will be decomposed into k tuples, which are stored in `RU`: $(o, A_1, v_1), \ldots, (o, A_k, v_k)$. A partial match query of the form (2) can be answered exactly by the following SQL query:

```
SELECT $x_1$.oid
FROM   RU $x_1$, RU $x_2$, ..., RU $x_p$
WHERE  $x_1$.Oid = $x_2$.Oid and
       ...
       $x_1$.Oid = $x_p$.Oid and
       $x_1$.Attr = '$B_1$' and $x_1$.val = '$w_1$'
       and ... and
       $x_p$.Attr = '$B_p$' and $x_p$.Val = '$w_p$'
```

To make this technique practical we build two indexes for the table `RU`, a clustered index on `(Attr, Val)`, and an unclustered index on `Oid`. If one of the p predicates is very selective, the optimizer has a very efficient plan: start by looking up the selective predicate in the `(Attr, Val)` index, then index-join the result with the other $p - 1$ tables. But if none of the predicates is selective, then there are no efficient plans. An alternative to the `RU` table is a universal table, where each data item is a tuple with lots of null values. This, however, is not a feasible representation because of two reasons. First, according to the data model we have assumed, the complete set of attributes is not know a priori. Secondly, even if this is known, a large set of attributes will make the size of the universal table prohibitively large. In fact, some commercial database systems impose an upper bound on the number of attribute allowed in a table.

Example 2.2 (cont'd) Continuing Example 2.1, the partial match query with four predicates gets translated into a four way join. The query took 304453 logical[2] disk I/O's to run on a SQL Server database to find the 43 objects in the answer.

The False Positives Approach An alternative approach is the following. Pick some $i = 1, \ldots, p$, and execute the following SQL query:

```
$Q_i$: SELECT x.Oid
       FROM   RU x
       WHERE  x.Attr = '$B_i$' and x.Val = '$w_i$'
```

This query translates into a single lookup in a clustered index, which is very efficient. The answer to Q_i contains all answers to the partial match query q, plus false positives. We need to eliminate these in a separate step, by iterating over all resulting `Oid`'s and checking the other $p - 1$ predicates[3]. We can optimize this: using data statistics, pick that i for which $SUP_U(Q_i)$ is smallest. Or, if data statistics are not available, then execute all queries, $Q_1, \ldots, Q_p$ (they can all be run in the same stored procedure), and return the result with the smallest

[2]The number of logical disk I/O's reported by the SQL Server is the total number of accesses to disk pages, regardless of whether the pages are in the buffer cache or not. It is a more accurate measure of the query's complexity than the running time.

[3]This approach assumes that objects can be accessed efficiently by their oid.

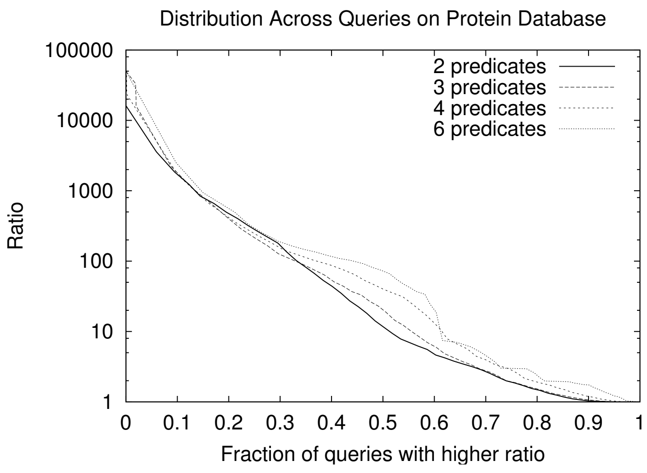


Figure 1: The ratio $|SUP_U(p_i)|/|SUP_U(q)|$ for various queries q.

cardinality[4]. Again, this approach works well when one of the predicates is very selective. Otherwise, the post-processing step becomes quite inefficient.

Example 2.3 (cont'd) Continuing Example 2.1, the partial match query can be answered in two steps. First, answer the query `feature-type='domain'`, which returns 35,004 items. This query took only 259 logical disk I/O's to run on SQL Server. Next we need to scan over them to find the 43 items that match the entire query. Unfortunately, there are no efficient solutions for this step, other than a sequential scan over the 35,004 objects. Fetching these objects takes at least an additional 35,004 logical disk I/O's.

Discussion An efficient solution to the partial match indexing problem should answer a query q in time proportional to $|SUP_U(q)|$. Instead, the approaches described above take time at least proportional to $|SUP_U(p_i)|$, where p_i is the most selective predicate in q. The question is how likely are queries for which $|SUP_U(p_i)| \gg |SUP_U(q)|$, like the query illustrated in Example 2.1, where the gap was three orders of magnitude. We argue that they are quite likely. The analogy in Web search engines is when users type multiple keywords: this usually happens after they tried single keywords and received too many answers. Anecdotal evidence we collected from scientists interested in sharing data through peer data management systems is that they would need queries consisting of several predicates, say 4 to 6 or more, in order to narrow their search to only few data sources: no individual predicate is selective enough to narrow the search. Finally, in a more systematic investigation, we analyzed the Protein Sequence Database in Example 2.1 and inspected all possible queries with 2, 3, 4, and 6 predicates that return some non-empty answers on the data set. For each of them we computed the fraction f between the number of items matched by the most selective predicate and the number of items matching the entire query. For example, for the query in Example 2.1 this number would be $f = 35,004/43 = 814$, because the most selective predicate matched 35,004 objects, while the entire query matched 43 objects. The graph in Figure 1 shows the cumulative number of queries that have f above a certain value. Reading from left to right on the x axis, 20% of the queries had $f \geq 625$, 40% of the queries had $f \geq 45$, etc. In other words, if one chooses a query at random, with a probability of 0.2 the value f is ≥ 625. Thus, queries for which $|SUP_U(q_i)| \gg |SUP_U(q)|$ exists, and form a non-negligible fraction of the set of all queries.

[4]One may be tempted to intersect the set of oid's in the p answer sets, but this brings us back to the join approach.

Other Approaches to Partial Match Queries Bitmaps are an effective way to perform intersections. Several database systems support bitmap indexes. The problem here is that, when used for partial match queries, the bitmaps take in the worst case time that is proportional to the number of objects matched by each predicate. Thus, their runtime is proportional to $\sum_i |SUP_U(p_i)|$, and not to $|SUP_U(q)|$. Compression techniques can speed up execution, and could prove effective in certain cases in practice. But they lack theoretical guarantees. In this paper we focus solely on techniques that have strong time guarantees, and do not consider compressed bitmaps.

Early approaches to the partial match problem [4] were designed in the context of a single table, with a fixed and not too large set of attributes. They are not appropriate for heterogeneous data.

3 The Partial Match Index (PMI)

We describe here our Partial Match Index, PMI. It is implemented on top of a relational database, and consists of only a few tables, each with an associated clustered index. The PMI has two parameters, chosen by the systems administrator to adjust the time/space trade off: S, a natural number, and $\varepsilon > 0$. The number of logical disk I/O's needed to answer a query q is guaranteed to be $\leq \max(S, (1+\varepsilon)|SUP_U(q)|)$, plus a negligible overhead.

The Tables PMI consists of the following tables:

- The table `RU(Oid, Attr, Val)` was introduced in Sec. 2.2 and has a clustered index on `(Attr, Val)`.
- The table `RU'`, which is a copy of `RU`, but with a clustered index on `Oid`.
- A number of tables L_1, L_2, ..., L_k, ...The exact number of tables depends on the data instances, but is typically small (say, under 10 or so). For each $k \geq 1$, L_k has the schema L_k(`Oid, Attr`$_1$`, Val`$_1$`, ..., Attr`$_k$`, Val`$_k$`)`. An entry $(o, A_1, v_1, \ldots, A_k, v_k)$ in L_k denotes the fact that the object o contains all attribute values pairs (A_i, v_i), $i = 1, \ldots, k$, and possibly more. In other words, o is in the answer to the query $q = [A_1 = v_1, \ldots, A_k = v_k]$. The table L_k will contain *all* answers o to *some* queries q. We will explain below which queries are selected to be included in L_k. To eliminate redundancies, all tuples $(oid, A_1, v_1, \ldots, A_k, v_k)$ in L_k have the predicates ordered lexicographically, i.e. $(A_1, v_1) < (A_2, v_2) < \ldots < (A_k, v_k)$. By definition $(A, v) < (B, w)$ if $A < B$, or $A = B$ and $v < w$.

We first describe how to search in the PMI, then define which queries are stored in the level k tables.

Answering Queries with the Stratified Index Consider a query $q = [B_1 = w_1, \ldots, B_p = w_p]$. We first compute a set of oids denoted $find(q)$, which contains all answers to q plus some false positives, and $|find(q)| \leq \max(S, (1+\varepsilon)|SUP_U(q)|)$. Then we iterate over all oids in this set and eliminate the false positives.

The function $find(q)$ is computed as follows. Assume that $(B_1, w_1) < (B_2, w_2) < \ldots < (B_p, w_p)$. For each $k = 1, \ldots, p$, let $q_k = [B_1 = w_1, \ldots, B_k = w_k]$ be the subquery consisting of the first k predicates in q. For each $k = p, p-1, \ldots, 1$, lookup q_k in L_k. If we find it, then $find(q)$ is the corresponding set of oid's from L_k (this is precisely $SUP_U(q_k)$, hence it contains all answers to q plus some false positives). Otherwise we continue with $k-1$, etc. If we reach $k = 1$ and still couldn't find $q_1 = [B_1 = w_1]$ in L_1, then we search q_1 in `RU` and return the set of oids from `RU`. The number of logical disk I/O's to compute $find(q)$ is negligible: it consists of at most p unsuccessful index lookups, followed by one successful lookup, which returns $|find(q)|$ oid's

To eliminate the false positives, we iterate over all oid's in $find(q)$, and check the remaining predicates $B_i = w_i$, for $i = k+1, \ldots, p$, by looking up that object in `RU'`. This step takes $|find(q)| \leq \max(S, (1+\varepsilon)|SUP_U(q)|)$ logical disk I/O's (assuming that each object fits on one page) because of the clustered index on `Oid`.

Defining L_k The difficult problem is to decide which queries to store in L_k: if we choose to include all queries, then the size of the entire PMI is prohibitively large. We start by including in L_1 all unary queries q for which $|SUP_U(q)| > S$. That is, L_1 contains only those (attribute, value)-pairs with high support. Next we build the tables $\mathtt{L}_2$, $\mathtt{L}_3$, $\ldots$, in this order, as follows. To build $\mathtt{L}_k$, we consider all queries with k predicates that occur in the data instance U: $q = [A_1 = v_1, \ldots, A_k = v_k]$, and decide whether to insert q in $\mathtt{L}_k$ or not. By "inserting q in $\mathtt{L}_k$" we mean inserting all tuples $(oid, A_1, v_1, \ldots, A_k, v_k)$, for $oid \in SUP_U(q)$. To make that decision, we check the following condition. Assume that $(A_1, v_1) < (A_2, v_2) < \ldots < (A_k, v_k)$, and consider the query $q_{k-1} = [A_1 = v_1, \ldots, A_{k-1} = v_{k-1}]$. Compute the set $Answ = find(q_{k-1})$: this is the approximate answer that we would obtain if we try to answer q, and q were not in $\mathtt{L}_k$. Notice that it is possible to compute $find(q_{k-1})$ at this stage because all tables below $\mathtt{L}_k$ are already computed. We insert q in $\mathtt{L}_k$ if $S <\mid Answ \mid$ and $(1+\varepsilon) \mid SUP_U(q) \mid < \mid Answ \mid$. To see why, consider what happens if we don't insert it. In that case $find(q) = Answ$. Clearly $SUP_U(q) \subseteq Answ$, but the question is how many false positives do we have. We know that either $\mid Answ \mid \leq S$, or $\mid Answ \mid \leq (1+\varepsilon) \mid SUP_U(q) \mid$, hence $find(q) \leq \max(S, (1+\varepsilon)|SUP_U(q)|)$.

This completes the definition of the PMI. Assuming that each object can be stored on one page, the following holds:

Theorem 3.1. *The number of logical disk I/O's needed to answer a partial match query q with the PMI is $\leq \max(S, (1+\varepsilon)|SUP_U(q)|)$, plus a negligible amount.*

Example 3.2 (cont'd) Continuing example 2.1, we answer the query q by issuing the following query[5]:

```
SELECT x.Oid
FROM   L4 x
WHERE  x.Attr1 = 'author' and x.Val1 = 'White,_'
and    x.Attr2 = 'citation' and x.Val2 = 'nature'
and    x.Attr3 = 'feature-type' and x.Val3 = 'domain'
and    x.Attr4 = 'year' and x.Val4 = '2000'
```

(Notice that we ordered the attributes lexicographically.) This query returned the 43 answers directly, and used only 5 logical disk I/O's on SQL Server. Recall that the join-approach took 304453, while the false-positives approach took $259 + 35004$ logical disk I/Os.

We defer a discussion on the size of the PMI to Sec. 4. Here, we discuss first how to build the PMI.

3.1 Construction of PMI

The definition of the PMI cannot be turned into a construction procedure, because it enumerates all queries in the data, which is too inefficient. The technique we use to build the PMI is somewhat similar to the item set generation algorithm in data mining. We assume that the data instance U is available in a file and can be read sequentially.

We build the indexes iteratively. For each k, the construction of L_k requires exactly two scans over U. At each level, we create two intermediate tables:

- CNT_k is the count table and it stores all k-predicate queries which go into L_k along with their count.
- CND_k is the candidate table and it stores all k-predicate queries q such that for any $i \leq k$, the query formed by taking the first i predicates of q does not belong to CNT_i with count less than S.

The queries in CND_k are the only candidates for insertion into L_k. This is because if some L_i stores a query q with support less than S, none of its extensions have to be considered in higher level indexes, as all of them can be answered using q.

[5]We omit the complete paths and show only the last tag. In practice, we assigned a separate identifier to each of the 75 distinct paths in our data set and used that instead.

$q_1 : (p_1, p_2, p_3, p_4)$
$q_2 : (p_1, p_2, p_3, p_5)$
$q_3 : (p_1, p_5, p_6, p_7)$
$q_4 : (p_2, p_5, p_6, p_8)$
$q_5 : (p_2, p_5, p_8, p_9)$

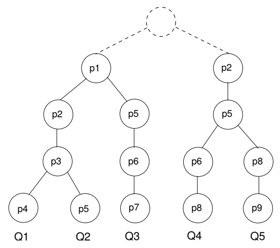


Figure 2: Sample Queries and Query-Tree

Now we describe how to build the level k index, having constructed all lower level indexes.

CND_k is initialized to the subset of $CND_{k-1} \times CND_1$ for which the $(k-1)^{th}$ predicate is smaller than the k^{th} predicate (because we only store queries in lexicographical order). By the definition of CND_k, any query not in CND_{k-1} cannot be the initial part of a query in CND_k. This significantly reduces the number of queries we need to consider for L_k.

Now we construct the count table. Note that since CND_k only contains a set of queries and not their support, its size is small compared to the data. We load the CND_k table into the main memory. Then, we perform a single scan over U and with each data item read, we update the counts of all the queries that contain it.

To speed up this computation, we create a tree representation of CND_k in memory. Every node in the tree is a predicate and every leaf corresponds to a query. Figure 2 shows few queries belonging to CND_4, where each p_i denotes a predicate, and the corresponding tree.

For each data item read, the tree is searched from top to bottom. If the data item doesn't satisfy some node, the whole subtree below the node is pruned. The query-tree representation not only results in sharing of computations(testing predicates) between queries, it also compresses the size of CND_k.

All candidate queries along with their counts are inserted in CNT_k. Next we prune the count table. We remove a query $q = (a_1, v_1, \cdots a_k, v_k)$ from CNT_k if there exists a query $q_i = (a_1, v_1, \cdots a_i, v_i)$ in CNT_i with count less than $(1+\varepsilon)$ times the count of q. We are left with precisely those queries in CNT_k that need to be inserted in L_k. We load back CNT_k into memory as we did for CND_k, and perform another scan over U and insert into L_k data items along with the matching queries. Finally, we prune the CND_k table by removing those queries which occur in CNT_k with count less than S.

3.2 The Cut-off Optimization

In some applications queries that return too many answers do not have to be answered exactly. If a query returns too many answers, users are likely to refine their query (by adding more predicates) and are not interested in seeing all the answers. We can exploit this in the PMI and save space by removing queries with large support.

Let Q_u be a number representing the threshold after which exact answers are no longer interesting. There are two ways to implement this optimization:

1. **Hard Cut-off** If a user issues a query q s.t. $|SUP_U(q)| > Q_u$, then an error message is returned.

2. **Soft Cut-off** If the user issues a query q s.t. $|SUP_U(q)| > Q_u$, then only Q_u objects need to be returned, together with a warning message.

To chose the PMI parameters, we proceed as follows. Choose $S > Q_u$ to represent an upper bound on the cost that is acceptable to answer a query, and define $\varepsilon = S/Q_u - 1$. In this way, if the user issues a query q s.t. $|SUP_U(q) \leq Q_u$, then the number of objects inspected by the PMI is $\leq max(S, (1+\varepsilon)|SUP_U(q)|) \leq max(S, (1+\varepsilon)Q_u) = S$.

The PMI definition is then modified as follows. For the Hard Cut-off, do not store any queries in index with support more than S, because by definition of PMI, these queries are used to answer only those queries whose support is $> S/(1+\varepsilon) = Q_u$. If a user issues a query with support greater than Q_u, it is simply not answered. In the Soft Cut-off, for queries with support greater than S we store only S tuples chosen randomly. Again, this does not affect queries of size less than Q_u and they are answered exactly and efficiently. For queries larger than Q_u, the index only returns Q_u tuples (or more) instead of the complete answer.

3.3 Handling Updates

Let U be the data instance on which the index is constructed. Let ΔU be a set of data items to be inserted or deleted from the data instance. We describe a *soft update* to the index that takes time proportional to the size of ΔU. The resulting index answers queries on the new data correctly and provides performance gurantees similar to the old index provided ΔU is small compared to U. The soft update is carried out as follows.

- **Soft Insert** : Let ΔU be the set of data items to be inserted. For each level k, CNT_K is loaded into memory and ΔU is scanned. All data items that match any of the queries in CNT_k are inserted into L_k along with the matching queries.

- **Soft Delete** : Let ΔU be the set of data items to be deleted. For each level k, delete from L_k all data items contained in ΔU.

It is easy to see that in the new index, each query in the index stores correct answer. However, the index no longer provides any performance gurantees as ΔU may change the ration of the sizes of a query and its subquery. This will not be significant for small updates. The performance can progressively detoriate as more and more updates occur. To restore the performance, the index can be reconstructed from the new data.

To summarize, changes in the data instance can be handled using quick soft updates, along with infrequent rebuilds when the current data becomes significantly different from the original data.

4 Analysis and Variations

We now discuss and analyze the Partial Match Index (PMI) theoretically and place it in the context of related work done on the set containment problem. Our discussion will also lead us to alternative indexes for the partial match problem, which we will compare experimentally to PMI in Sec. 5.

4.1 Background: The Set Containment Problem

An extensively studied problem that is equivalent to the partial match problem is the *set containment problem*. Let $[m] = \{1, 2, \cdots m\}$. An object s is a subset of $[m]$, $s \subseteq [m]$. As before we associate an oid to each object, and denote with U a data instance, i.e. a set of objects. A *query* is also a subset of $[m]$. The answer q consists of all objects in U that contain q. We also call this set the *support* of q, $SUP_U(q) = \{s | s \in U, q \subseteq s\}$. Notice that $q \subseteq q'$ implies $SUP_U(q) \supseteq SUP_U(q')$ and if $q = \emptyset$ then $SUP_U(q) = U$.

We next define the set containment problem.

Problem 2. *(Set Containment Indexing Problem) Given a data instance U construct a data structure (the index) s.t. for every query $q \subseteq [m]$, the answer $SUP_U(q) = \{s_1, \ldots, s_k\}$ can be computed efficiently. We will denote $n = |U|$ the number of objects in the data instance.*

The partial match problem and the set containment problem can easily be reduced to each other as follows. Consider a partial match problem, and number all possible attribute-value pairs occurring in the dataset from 1

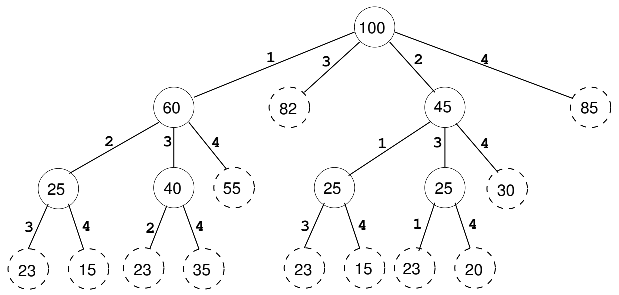


Figure 3: Illustration for CIP

to m. Each object can be thought of as a set consisting of attribute-value pairs, and hence, a subset of $[m]$. A partial-match query also specifies a set of attribute-value pairs and again can be represented by a subset of $[m]$. The answer to the query is precisely the set of data items containing the query as a subset. The converse reduction is trivial.

The time/space trade off of the set containment problem has been studied theoretically. One extreme approach is answer each query with a sequential scan over the data: here the time complexity is linear in the size of the data, and so is the space complexity. The other extreme is to precompute and store the answers to all 2^m queries: here the time complexity is sub-linear in the size of the data, but the space complexity is exponential. Until recently it was not clear that one can do any better than these two extremes.

Charikar, Indyk and Panigrahy [7] found the first index that has a better time/space compromise. We review it here, introducing our own notations. We call this algorithm CIP.

CIP has a parameter c, that can be adjusted to tune the time/space trade off. The index consists of a tree obtained follows. Each node corresponds to some query q and stores $SUP_U(q)$: multiple nodes may correspond to the same query. A node corresponding to q has children corresponding to queries of the form $q \cup \{p\}$ for $p \in [m], p \notin q$. The tree is constructed recursively as follows:

1. The root stores $SUP_U(\emptyset)$ (which is U).

2. A node x storing a set $SUP_U(q)$ has the following children. For each $p(1 \leq p \leq m)$, x has a child storing $SUP_U(q \cup \{p\})$ if $p \notin q$ and the following condition is true:

$$|SUP_U(q)| \geq |SUP_U(q \cup \{p\})| + \frac{n}{c}$$

   The edge is labeled as p. If the node is not stored, then we say that it is "pruned": none of its descendants are stored in this case.

Figure 3 shows a hypothetical tree constructed according to the CIP algorithm. There are four attributes ($m = 4$) numbered from 1 to 4. The numbers in the circle denote the size of the support of the corresponding query. Dotted circles denote the nodes that were pruned.

Given a query q, its answer is computed by the following nondeterministic algorithm. The tree is traversed starting from the root. If some node x storing $SUP_U(q')$ is reached, then we inspect whether x has at least one outgoing edge labeled p s.t. $p \in q$. If so, one such edge is chosen nondeterministically and the search continues with that child. Otherwise, $SUP_U(q')$ is returned: this may contain false positives, which are eliminated in one linear scan.

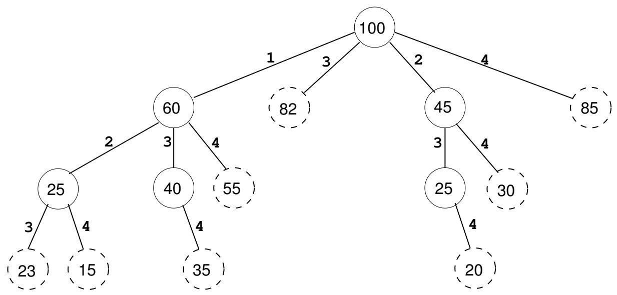


Figure 4: Illustration for CIP2

Space Complexity: Each node stores at least $\frac{n}{c}$ fewer objects than its parent. Also, the root stores n objects. Thus, the depth of the tree is bounded by c. Also, the maximum fanout is m. So there are $O(m^c)$ nodes in the tree, each storing at most n objects. Thus, the space taken by the algorithm is $O(nm^c)$.

Time Complexity: Suppose the search ends at node x. For each $p \in q$, there can be at most $\frac{n}{c}$ objects stored at x that do not contain p. So there can be at most $\frac{mn}{c}$ objects at x that do not contain q and need to be filtered. So the time taken by the algorithm is $O(c + |SUP_U(q)| + \frac{mn}{c})$. Notice that this upper bound holds for every nondeterministic choice of the algorithm.

4.2 Optimizations to CIP

In its original formulation CIP cannot be used as a practical algorithm for the partial match problem. We introduce here several optimizations and variations to make practical.

Order Optimization In CIP, some query results may be stored more than once. For instance, a query $\{1, 2\}$ may be stored after expanding the edge 1 followed by 2 as well as expanding 2 followed by 1. In general, a query with k elements can possibly occur k! times. The CIP algorithm can be modified to remove the storage of duplicate queries. The following rule is added while expanding a node.

Rule 1. *Expand the tree in a breadth-first fashion. Let x be the node to be expanded on behalf of query q. Let the path from the root to x go through nodes $x_1, x_2 \cdots x_k$ and edges labeled $p_1, p_2 \cdots p_k$ (the edge p_k is from x_k to x; notice that $q = \{p_1, \ldots, p_k\}$). Do not expand x if there exists i such that $p_k < p_i$ and x_i has an outgoing edge labeled p_k.*

Figure 4 shows the tree in Figure 3 with the order optimization. To see an instance of Rule 1 consider the node labeled 25 in Fig. 3 reachable by 2 then 1. Its ancestors are labeled 100 and 45. The node 100 has already an outgoing edge labeled 1 (to 68), and $1 < 2$, hence 25 is eliminated.

The search is modified as follows, and made deterministic: at each node x, the smallest p is traversed such that query contains p and x expands p.

Lets call this algorithm as CIP2. We prove:

Theorem 4.1. *In CIP2 algorithm, no query is stored twice.*

Proof. On the contrary, let us assume that there is such a query q. Let x_i and x_j be the nodes that store q, and let x be their least common ancestor. Let p_i and p_j be the edges from x leading to x_i and x_j respectively. We can assume $p_i < p_j$ w.l.o.g. By Rule 1, no descendant of x reached by p_j can expand p_i. So, it is impossible for x_j to store q, leading to a contradiction. □

Theorem 4.2. *The CIP2 algorithm has the same time bounds as CIP.*

Proof. Let q be the given query. After following the search algorithm, suppose we reach a node x that cannot be traversed further. We will show that there cannot be an edge p such that $p \in q$ and p is not expanded because of Rule 1. This will show that CIP2 and CIP have the same time complexity. So, on the contrary, let us assume that there is such an edge p. This implies that some parent x_i of x expands p. Let p_i be the edge that leads to x. By Rule 1, $p < p_i$. So the search algorithm will take the edge p at x and never reach node x, which leads to a contradiction. This proves the theorem. □

Rule 1 is equivalent to the following simpler rule. Rule 2 is faster to enforce than Rule 1 as we do not need to check all the parents of a node before expanding it.

Rule 2. *Expand the tree in a breadth-firth fashion. Let x be the node to be expanded with incoming edge labeled p. Let x' be its parent, and let its incoming edge be labeled p'. Do not expand x if $p' > p$.*

Rule (2) enforces a simple structure on the tree. Let $p_1, p_2, \ldots, p_k$ be the labels on the edges on the path from the root to some node x. Then $p_1 < p_2 < \ldots < p_k$.

The equivalence of the two rules follows easily from the following lemma.

Lemma 4.3. *In algorithm CIP, it a node x doesn't expand p, none of the ancestors of x expands p.*

The lemma is a direct consequence of the fact that number of data items at x not containing p is less that the number of data items not containing p at any ancestor of x.

Algorithm MULT In CIP2 the the running time for a query q is not guaranteed to be proportional to $|SUP_U(q)|$. The following change can provide this guarantee, and also motivates the construction of the PMI below: replace the additive condition for expanding the node in the CIP algorithm by a multiplicative condition. For that we introduce a new parameter $\varepsilon > 0$. A node x storing a set $SUP_U(q)$ has a child corresponding to query $q \cup \{p\}$ if the following is true:

$$|SUP_U(q)| \geq |SUP_U(q \cup \{p\})| + \varepsilon |SUP_U(q)|$$

Time Complexity: Let q be the query to be answered, and consider a node x (corresponding to some query q') that does not expand any of the elements in q. There are $|SUP_U(q')|$ objects stored at x. Therefore, given any $p \in q$, there are at most $\varepsilon|SUP_U(q')|$ objects stored at x that do not contain p. In all, there are at most $\varepsilon m|SUP_U(q')|$ sets not containing q. So time taken by the algorithm is $\frac{1}{1-m\varepsilon}|SUP_U(q)|$.

Space Complexity: The maximum fanout is m and the depth is $\frac{\log n}{\log 1/(1-\varepsilon)}$. So the space required is $O(nm^{\frac{\log n}{\log 1/(1-\varepsilon)}})$.

The Rule 1 can be applied to MULT as well to achieve the order optimization [6].

4.3 Algorithm PMI

Finally, we can express our Partial Match Index PMI described in Sec. 3 in a way in which it can be analyzed and compared with the variations on CIP described here.

Recall that PMI takes two parameters, S and ε. PMI also builds a tree, with each node storing $SUP_U(q)$ for some query q. We apply Rule 2, hence the association from nodes to queries is one-to-one, and each path in the tree has increasing edge labels. We prune nodes according to a multiplicative criteria. But unlike the previous algorithms, a node can be pruned while some of its descendants are not pruned. We call a node *alive* if it is not pruned.

Algorithm PMI The tree is constructed breadth-first:

[6] Its simplified version, Rule 2, is not applicable for MULT

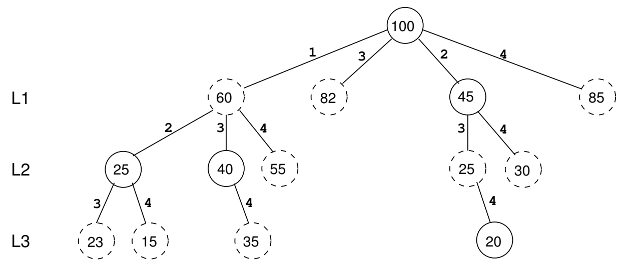


Figure 5: Illustration for PMI

1. The root is associated to the query $q = \emptyset$ and stores $SUP_U(\emptyset)$ (which is U). It has m children, associated to the queries $\{1\}, \ldots, \{m\}$ respectively; the edges entering these children are labeled $1, \ldots, m$ respectively.
2. Let x be a non-root node, associated to some query q, and with incoming edge labeled p ($p \in q$). Then x has children associated to $q \cup \{p'\}$, for $p' > p, p' \notin q$. (This enforces Rule 2.) The child associated to $q \cup \{p'\}$ has incoming edge labeled p'.
3. If node x associated to query q has some live ancestor x' associated to q' s.t. either (a) $|SUP_U(q')| \leq S$ or (b) $|SUP_U(q')| \leq (1+\varepsilon)|SUP_U(q)|$, then x is pruned and $SUP_U(q)$ is not stored. Notice that its descendants are not necessarily pruned. If condition (a) holds for x then it also holds for all its descendants and these are pruned too; if condition (b) holds then some of the descendants may be alive. If neither conditions (a) or (b) hold then x is alive, and $SUP_U(q)$ is stored at node x.

Given a query $q = \{p_1, \ldots, p_k\}$ its answer is computed by the following algorithm. Assume $p_1 < p_2 < \ldots < p_k$, and let x be the node associated to the query q (x is reachable from the root by following the edges labeled $p_1, p_2, \ldots, p_k$ in this order). Starting from x, follow the path to the root until the first node x' which is alive (x' is x if x happens to be alive). Return $SUP_U(q')$, where q' is the query associated to x': this may contain false positives, which are eliminated in a linear scan.

Figure 5 shows the same tree as 3 with algorithm PMI. In the actual data structure the pruned nodes need to be eliminated, and their children promoted.

The connection to the PMI relational structure described in Sec. 3 is the following. L_k contains precisely the queries q corresponding to the live nodes at depth k in the tree.

Time Complexity: It takes $O(m)$ time to find the node x corresponding to the given query q. The set $Answ = SUP_U(q')$ that is actually returned satisfies $|Answ| \leq \max(S, (1+\varepsilon)|SUP_U(q)|)$. Hence the running time is bounded by $O(m + \max(S, (1+\varepsilon)|SUP_U(q)|))$.

Space Complexity: To analyze the space complexity, we proceed in two steps. First we express the space as a function of the number of live-leaf nodes, then estimate the number of live-leaf nodes.

A *live-leaf node* is a node x which is alive, and which has no live descendants. If one collects all live nodes in the PMI tree then the live-leaf nodes form the leaves of this subtree. Let the live-leaf node x be associated to a query q; then:

$$|SUP_U(q)| \quad \leq \quad S \tag{3}$$

Moreover, let x_0 be the parent of x in the PMI tree (x_0 is not necessarily live), and $q_0 = q - \{\max(q)\}$ be the query associated to x_0. Then:

$$|SUP_U(q_0)| > S/(1+\varepsilon) \tag{4}$$

Indeed, suppose not. Then, in particular x_0 is not alive (otherwise it would be a live-leaf node, contradicting the fact that x is alive). Let x' be the first live ancestor of x, with associated query q'. We have $|SUP_U(q')| > S$ and $|SUP_U(q_0)| \leq S/(1+\varepsilon)$, which implies that $|SUP_U(q') > (1+\varepsilon)|SUP_U(q_0)|$, contradicting the fact that x_0 is not alive. Thus, we have shown that live-leaf nodes satisfy Equations (3) and (4).

We first prove:

Proposition 4.4. *Let L be the total number of live-leaf nodes in a PMI tree. Then the total number of live nodes is $\leq L(2 + \frac{\log(n/S)}{\log(1+\varepsilon)})$.*

Proof. Equation (4) says that the parent of a life-leaf node x contains at least $S/(1+\varepsilon)$ data objects. Each live ancestor of x will have at least a factor of $(1+\varepsilon)$ more data objects than the previous live ancestor. Hence the total number of live ancestors of x, including itself, is $\leq 1 + \log_{1+\varepsilon}(n(1+\varepsilon)/S) = 2 + \frac{\log(n/S)}{\log(1+\varepsilon)}$. This proves the proposition. □

We now examine the number of live-leaf nodes, denoted L. Rather than giving a general upper bound, we consider a probability distribution on the attributes in $[m] = \{1, 2, \ldots, m\}$, and compute the expected value for L. Let $pr(p)$ denote the probability that the attribute p belongs to a data object. In other words, the expected number of data objects d that contain p ($p \in d$) is $n \times pr(p)$. Given a query $q = \{p_1, \ldots, p_k\}$, we denote $pr(q) = \prod_{i=1,k} p_i$. Assuming the probabilities of the attributes to be independent, the expected value of $|SUP_U(q)|$ is $e(q) = n \times pr(q)$. Given a query q and a number $S \leq n$, we denote with $pr(q, S)$ the probability that $|SUP_U(q)| > S$. We have:

$$pr(q,S) = pr(|SUP_U(q) > S) \quad = \quad \sum_{t=S+1,n} \binom{n}{t} (pr(q))^t (1 - pr(q))^{n-t}$$

We need an upper bound for $pr(q, S)$, and for that we use the following:

$$\binom{n}{t} \quad = \quad \binom{n-S}{t-S} \frac{\binom{n}{S}}{\binom{t}{S}} \leq \binom{n-S}{t-S}\binom{n}{S}$$

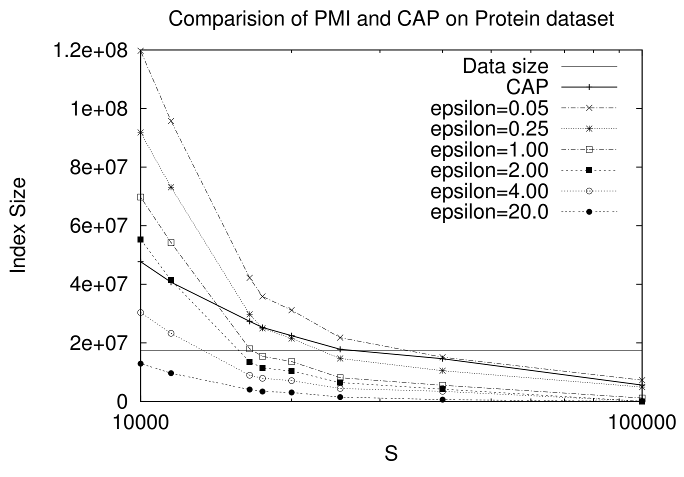
which implies:

$$pr(q,S) \quad \leq \quad (\binom{n}{S} pr(q)^S) \sum_{t=S+1,n} \binom{n-S}{t-S} pr(q)^{t-S} (1-pr(q))^{n-t} < \binom{n}{S} pr(q)^S < (e(q))^S$$

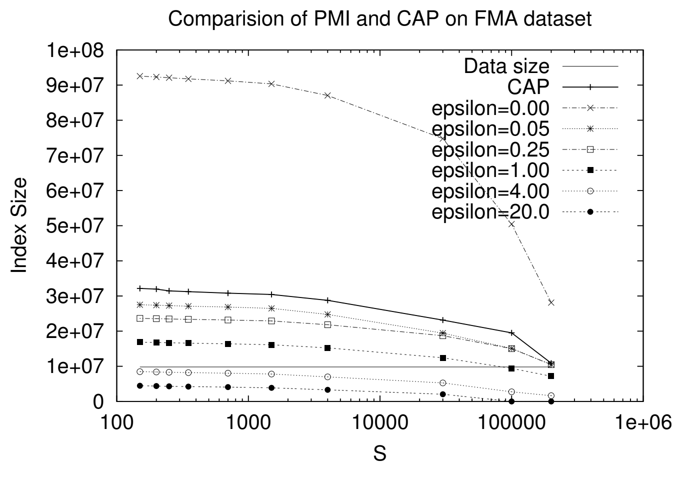
Recall that $e(q)$ is the expected size of the answer of q.

Now we will compute the expected number of live-leaf nodes L in the PMI tree. Let x be a node associated to the query q. The probability that it is a live-leaf node follows from conditions (3) and (4) which characterize live-leaf nodes. Namely, this probability is $pr(q,S)(1 - pr(q_0, S/(1+\varepsilon)))$, where $q_0 = q - \{\min(q)\}$. Hence the expected number of live-leaf nodes is bounded by:

$$\begin{aligned} L &= \sum_{q \subseteq [m]} pr(q_0, S/(1+\varepsilon))(1 - pr(q,S)) \\ &\leq \sum_{q \subseteq [m]} pr(q_0, S/(1+\varepsilon)) \\ &\leq m \sum_{q_0 \subseteq [m]} (pr(q_0, S/(1+\varepsilon))) \\ &\leq m \sum_{q_0 \subseteq [m]} (e(q_0))^{S/(1+\varepsilon)} \qquad (5) \end{aligned}$$



(a)



(b)

Figure 6: Index Sizes for CIP2 and PMI

So far we have not made any assumptions on the probability distributions of the attributes, $pr(p)$. When the distribution is uniform, i.e. there exists a constant c, $0 < c < 1$, s.t. $\forall p \in [m], pr(p) = c$, then expected value of $|SUP_U(q_0)|$ is $e(q_0) = nc^k$, where $k = |q_0|$, and Equation (5) becomes $mn(1 + c^{S/(1+\varepsilon)})^n$. For a Zipf distributions, $e(q_0) = nc^k/k$, for some constant c, $0 < c < 1$. For example, in the case of a simple Zipf distribution where $pr(p) = 1/p$, we have $e(q_0) = n \times \prod_{p \in q_0} 1/p \leq n \times 1/k! \approx n \times 1/(k \times e^{k/e})$. In this case Equation (5) becomes $m(1 + c^{S/(1+\varepsilon)})^n$, i.e. a factor of n less than the uniform case. In summary, we have:

Proposition 4.5. *The expected number of live-leaf nodes L is bounded by $mn(1 + c^{S/(1+\varepsilon)})^n$, in the case of a uniform probability distribution on the attributes, and by $m(1 + c^{S/(1+\varepsilon)})^n$ in the case of a Zipf distribution, where c is some constant,* $0 < c < 1$.

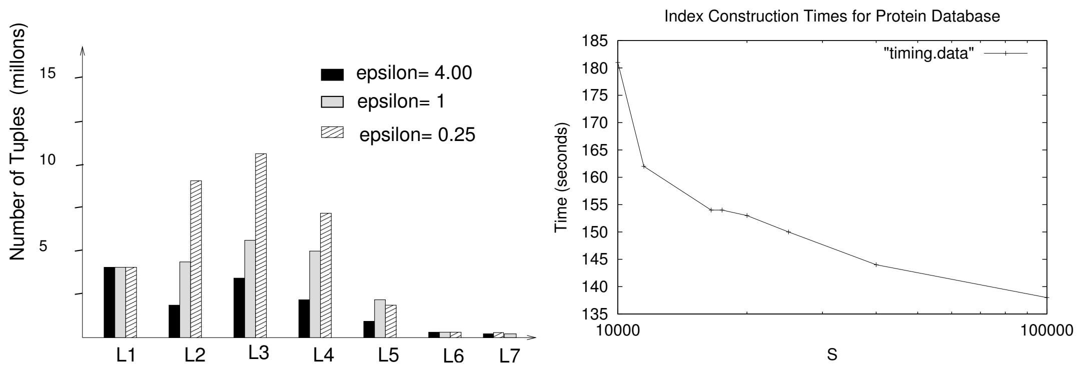
Together, Propositions 4.4 and 4.5 give us the space complexity of the PMI.

5 Experiments

We evaluated algorithms CIP2 and PMI on two datasets. Experiments were ran on a Linux 930 MHz processor with 256MB memory using C to construct the indexes and Microsoft SQL Server to execute queries on the indexes. The first dataset was the Protein Sequence Database (`http://pir.georgetown.edu/`), which consists of integrated collection of protein sequences. It has 75 distinct attributes. Not all data items have all the attributes and some data items have multiple values for the same attribute. There are around 260,000 data items with a total of 17.3 million attribute-value pairs and the total size of the database is 466 MB. The second database is called FMA and consists of the metadata of MRI scans of brain. It is 166 MB in size with 1.4 million data items. Each data item has exactly 7 attributes.

CIP2 uses a parameter c while PMI uses S. To compare them we chose c such as to give the same guarantees as PMI. For Protein database, c was S/10 as PMI stored queries with a maximum size of 10. For FMA database, c was put as S/7.

Figure 6 (a) shows the comparison of the sizes of indexes constructed using CIP2 and PMI on a log scale as a function of the parameter S. For PMI we potted several curves for various values of the parameter ϵ. For a fixed value of S, we observe that PMI performs better than CIP2 above a certain threshold value of ϵ beyond which



(a) (b)

Figure 7: Size of L_k ($S = 16500$) and time to construct the PMI (b). For Protein data.

CIP2 performs better. This threshold value decreases as S is increased. Figure 6 (b) shows the same graph for FMI. Again, PMI performs better that CIP2 above a fixed threshold value of ϵ. But, unlike Protein database, this threshold value does not increase as S increases. Both PMI and CIP2 algorithms seem to grow at the same rate. Secondly, this threshold value is very low. Even for $\epsilon = 0.05$, PMI outperforms CIP2 for all values of S.

Figure 7 (a) shows the sizes of L_k for the Protein database (it peaks at $k = 3$ and is zero after $k = 7$), while (b) shows the time needed to construct the index, excluding the time needed to load it into the relational database.

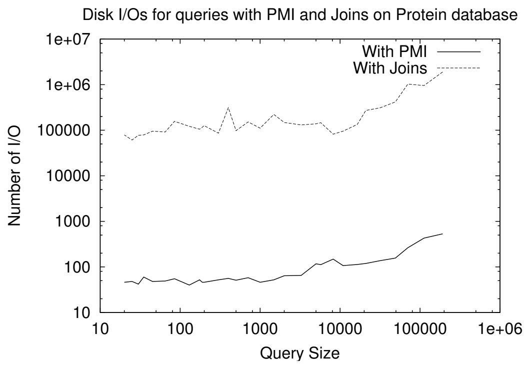
PMI was implemented on Microsoft's SQL server and we evaluated the efficiency of answering queries in terms of logical disk I/Os. We compared this with the number of I/Os required to answer same queries using the join-approach (Sec. 2.2). The results are shown in Figure 8 (a) and (b). Instead of reporting real time (which is distorted by factors beyond our control, such as cache hits etc) we report the total number of logical disk I/Os performed by the database, which is equal to the number of reads with cold cache. The comparison shows that queries using the naive method are around 9000 times slower than those using the PMI index on the Protein database.

Next we consider the effects of hard and soft cut-offs on the index size. Figure 9 (a) shows, on a log scale, the sizes of index on Protein database as a function of S with $\epsilon = 0.05$. We can see that soft cut-off results in a saving of 15 to 35% while hard cut-off reduces space by 40 to 60%. Figure 9 (b) shows the same graph for FMA database. For this database, with hard/soft cut in effect, the index size actually decreases as S decreases. This is due to the fact that with a decrease in S, the hard/soft cut makes the index more specialized to answer smaller queries. With Protein database, this phenomenon was not observed as it is a much larger database. With S being made sufficiently small, it would also show the same behavior but it would be prohibitively expensive to construct index on Protein database for such a small S.

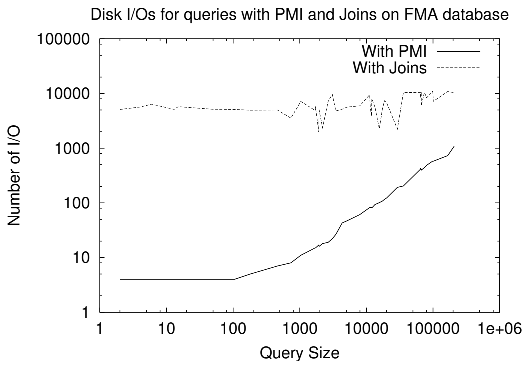
To summarize, we have experimental shown that PMI can be highly efficient in answering partial match queries, while consuming a only a practical amount of space.

6 Related Work

The earliest works on the partial match problem are the *k-d-trees* by Bentley[4], *quadtrees* by Bentley and Finkel[10] and *hashing and digital techniques* by Rivest[24]. All of these algorithms use space partitioning techniques. They work well only when the number of attributes m is small compared to the number n of data items



(a)



(b)

Figure 8: Efficiency of answering queries measured in logical disk I/Os.

($m \approx \log n$), and run in time $O(n^{(1-\frac{k}{d})})$, where k is the number of attributes specified in the query. In fact, Rivest conjectured that any algorithm with linear space would require $\Omega(n^{(1-\frac{k}{m})})$ time to answer such queries.

For a general n and m, and in particular, for high-dimensional data ($m \gg \log n$), there is no algorithm known that is fast and has small space requirements. Therefore, it is believed [6] that this problem suffers from the curse of dimensionality. Jayaram et al. [16] proved that in the cell probe model, any algorithm running in space polynomial in m must make $\Omega(m)$ probes. However, these lower bounds do not seem strong enough to justify the curse of dimensionality conjecture. Charikar et. al. [7] have recently given two algorithms that run in sub-linear time and sub-exponential space for large m (presented here in Section 4).

A related problem in multidimensional data is the nearest neighbors problem [5]. However, the approximate version of the nearest neighbor problem does not suffer from the curse of dimensionality [18]. No such analog is known for the partial match problem.

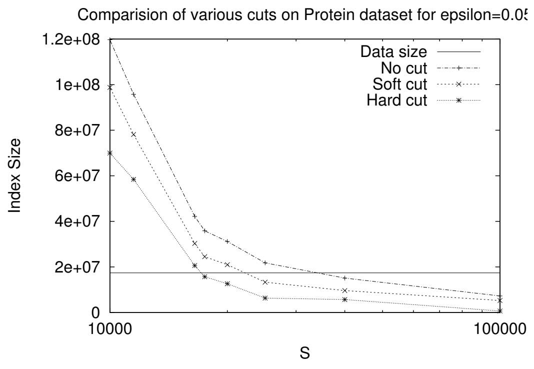
There has been work [2, 25, 32] on creating indexes to efficiently answer second order queries, i.e. queries which specify the values of two attributes. However, all of these techniques work in a naive way by considering all possible second order queries, dividing them into groups and storing the result for each group in a bucket. These techniques are applicable only when attributes take very few values and cannot be efficiently extended to higher order queries.

It has been shown [31] that most of the data structures based on space partitioning perform worse than linear scan for dimensions above 10. Several alternatives have been proposed to optimize the linear data scan like VA-files [29], signature based techniques [30] and bitmaps. In spite of these optimizations, the fundamental problem persists: the running time is proportional to the number of dataitems.

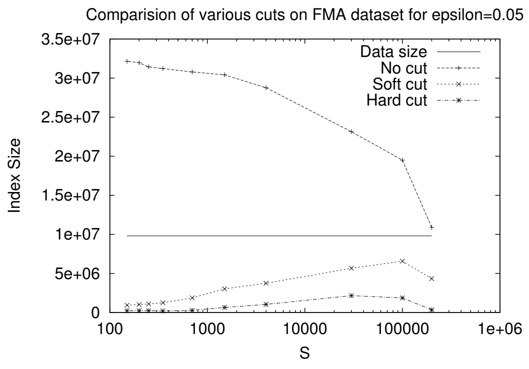
7 Conclusions

We have described a simple approach to searching heterogeneous data through *partial match queries*. We also described an efficient index structure for supporting partial match queries on heterogeneous data. The index can be implemented easily on any relational database engine, and provides guarantees on the number of physical disk I/O's performed to answer a query.

One interesting application of this index is to structured information sources on the Web. An issue for such an index is how to crawl the available sources and how to relate their schemas. We leave this for future work.



(a)



(b)

Figure 9: Hard and Soft Cut-offs

References

[1] Karl Aberer, Philippe Cudré-Mauroux, and Manfred Hauswirth. The chatty web: Emergent semantics through gossiping. In *International World Wide Web Conference*, Hungary, 2003.

[2] C. T. Abharam, S.P.Ghosh, and D.K.Ray-Chaudhuri. File organization schemes based on finite geometries. *Inform. Contr.*, 12(2):143–163, 1968.

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

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[5] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 142–153. ACM Press, 1998.

[6] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 312–321, 1999.

[7] M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *29th International Colloquium on Algorithms, Logic, and Programming*, pages 451–462, 2002.

[8] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.

[9] S. Dar, G. Entin, S. Geva, and E. Palmon. DTD's DataSpot: database exploration using plain language. In *VLDB*, pages 645–649, 1998.

[10] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Information*, 4:1–9, 1974.

[11] L. Galanis, Y. Wang, S. Jeffrey, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, pages 874–885, 2003.

[12] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proceedings of the Intenrnational Conference on the World Wide Web*, 2003.

[13] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management System. In *ICDE*, 2003.

[14] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, 2002.

[15] Institute for Systems Biology. http://www.systemsbiology.org.

[16] T.S. Jayram, Subhash Khot, and Ravi Kumar Yuval Rabani. Cell-probe lower bounds for the partial match problem. In *Annual ACM Symposium on Theory of Computing (STOC)*, 2003.

[17] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *SIGMOD*, pages 325–336, 2003.

[18] Jon M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Twenty-ninth annual ACM symposium on Theory of computing*, pages 599–608, 1997.

[19] Stephen H. Koslow. Should the neuroscience community make a paradigm shift to sharing primary data? *Nature Neuroscience*, 3(9):863–865, September 2000.

[20] Tanu Malik and Alex Szalay. Skyquery: A web service approach to federate databases. In *Proceedings of CIDR*, 2003.

[21] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, Cairo, Egypt, September 2000.

[22] Nature Neuroscience Editorial. A debate over fmri data sharing. *Nature Neuroscience*, 3(9):845–846, Sep 2000.

[23] NIH statement on sharing research data. *http://grants2.nih.gov/grants/policy/data_sharing/index.htm*, March 2002. U.S. National Institutes of Health.

[24] Ronald L. Rivest. Partial-match retrieval algorithms. *SIAM J. Comput.*, 5(1):19–50, 1976.

[25] S.P.Ghosh and C. T. Abharam. Application of finite geometries in file organization for records with multiple-valued attributes. *IBM J. Res. Develop.*, 12(2):180–187, 1968.

[26] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *SIGMOD*, pages 570–581, 2002.

[27] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems I.* Computer Science Press, Rockville, MD 20850, 1989.

[28] H. Wang, S. Park, W. Fan, and P. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[29] Roger Weber, Klemens Böhm, and Hans-J. Schek. Interactive-time similarity search for large image collections using parallel VA-files. *Lecture Notes in Computer Science*, 1923:83–??, 2000.

[30] Roger Weber, Klemens Böhm, and Hans-J. Schek. Interactive-time similarity search for large image collections using parallel VA-files. *Lecture Notes in Computer Science*, 1923:83–??, 2000.

[31] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 194–205, 24–27 1998.

[32] Sumiyasu Yamamoto, Shinsei Tazawa, Kazuhiko Ushio, and Hideto Ikeda. Design of a balanced multiple-valued file-organization scheme with the least redundancy. *ACM Transactions on Database Systems (TODS)*, 4(4):518–530, 1979.