Index Structures and Algorithms for Efficient Profile Matching

Nesime Tatbul

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-01-09
2001

Index Structures and Algorithms for Efficient Profile Matching

E. Nesime Tatbul

Brown University
Department of Computer Science
tatbul@cs.brown.edu

April 17, 2001

Abstract

In an environment like the web, where new data is continuously being generated and where users continuously need to receive new data, traditional data management techniques fall short. When users periodically poll the data sources for updates, unacceptably high server loads result. Publish/subscribe systems provide a solution to the polling problem by delivering relevant data to interested users as new data gets generated. Users declare the data they are interested in through long-term queries called *profiles*, and the system sends data to these users as new data gets generated.

A major problem in a publish/subscribe system is efficiently deciding which profiles match new data when there are a large number of profiles. In this research, we have investigated several index structures and profile matching algorithms to provide scalable operation of a publish/subscribe system. We have also analyzed the effect of overlap among profiles on the performance of the indices.

1 Introduction

Web has brought new opportunities and problems for both publishing and retrieving data. There is a huge amount of continually growing information on the web. The need to locate and access data also grows in parallel. Even if we assume that effective mechanisms to locate data exist, we still have the problem of getting immediate updates as new data of interest gets generated. Using traditional data management techniques, we can not do better than polling the data sources for new data. However, this is both cumbersome and leads to unacceptably high server loads. Publish/subscribe systems provide a solution to the polling problem by automatically delivering relevant data to interested users as new data gets generated. Users declare the data they are interested in through subscriptions and the system sends data to these users as it receives matching data from the data sources. Besides, the users need not worry about locating the relevant data sources any more.

Publish/subscribe systems are modeled in two major ways: group-based and content-based [FLPS00]. In group-based approach, data objects are grouped into subject categories and users subscribe to these groups to receive any data of that subject category. On the other hand, content-based approach allows the users to get subscribed to more specific data in terms of content. This way, users do not have to receive every data that is included in a broad subject category; they can be more selective. Subscriptions contain detailed specifications of content of data the users wish to receive. We call such specifications *profiles*. A profile can be thought of as a long-term query.

A major problem in a publish/subscribe system is efficiently deciding which profiles match new data when there are a large number of profiles. This problem is important to solve to achieve scalable operation of the system against increasing number of profiles and the rate of new data arrival. To be able to match profiles in an efficient way upon new data arrival, we need to perform some pre-processing on the profiles that will organize and prepare them for the matching step.

The traditional approach to matching individual queries to large amount of data is building indices on the data. In our case, the roles of queries and data are reversed. Now we have a large number of queries and we wish to match individual data items against those queries. Based on this analogy, we consider building indices on the queries to handle the profile matching problem.

The paper continues with presenting the related work in the next section. Later we present the basic system architecture and the data model that we consider in Section 3. The index structures and the matching

algorithms together with their performance evaluation are discussed in Section 4. Section 5 presents the contributions of our work. Finally, we conclude the paper by discussing the future work in Section 6.

2 Related Work

We can discuss the related work under three main headings:

- Rule evaluation in active databases
  A similar problem to the profile matching problem we are considering has previously been attacked in the context of active databases. The problem considered was efficient evaluation of rules/triggers (each consisting of predicates) upon the occurrence of an event.

  [HCKW90] proposes IBS (Interval Binary Search) tree structure to facilitate predicate matching in database rule systems. The focus of the work is on dynamism, i.e., making the tree easily-modifiable upon insertions and deletions of predicates. The tree should also be kept balanced not to lose efficiency. This tree structure resembles our Cluster Index. However, there are some differences in building methods. [HJ96] both provides a survey of selection predicate indexing algorithms and proposes a new indexing scheme for active databases based on Interval Skip Lists (IS-Lists). Dynamism issues for IS-Lists are also discussed. IS-Lists scheme is easier to implement but does not provide improvement over other existing methods in terms of evaluation performance. Finally, [OM97] provides a decision tree approach to filtering rules in trigger enabled databases. This tree ensures that each predicate is evaluated at most once, but the worst case is equivalent to the naive algorithm of evaluating all the predicates. They also allow disjunctions in triggering expressions. Dynamic addition and deletion of trigerring expressions is not handled. Moreover, the binary decision tree is not guaranteed to be the optimal one since this problem is known to be NP-complete. None of these works have considered join predicates.

- Selective dissemination of data
  There are two major related works in the area of selective dissemination of data: SIFT (Stanford Information Filtering Tool) [YGM94] and XFilter [AF00]. Both propose index structures and algorithms for efficient matching of documents against large number of profiles. However, the data models and the profile languages used are different.

  In SIFT [YGM94], a document is a collection of words and a profile is a sequence of distinct words. A profile matches a document if all words in the profile appear in the document. Inverted sets are used as the basis for the index structures. Three indexing methods are proposed: In the counting method, a hash table maps words to inverted sets of profiles that contain them. All distinct words in a document are applied to this structure and by counting the number of matching words, matching profiles are determined. Key method is a variation of the counting method in which a profile appears in the inverted list of one of its words. Finally, the tree method uses a trie-like structure and exploits the similarity of profiles by storing identifying prefixes of profiles in a tree structure instead of a hash table.

  XFilter system is a more recent system which aims at efficient filtering of XML documents [AF00]. In this system, profiles are represented as queries using XPath language which can specify path expressions over XML data. Documents are XML documents with schema hidden in tags rather than plain text documents. XPath enables profiles to refer to schema information in documents. XFilter uses a more sophisticated inverted index than SIFT's. For each profile, a Finite State Machine (FSM) whose states represent element tags in XPath query is constructed. Inverted set index is built over the states of the FSMs of profiles. An event-based XML parser is used to trigger the states in FSMs of profiles which in turn decide which profiles match the XML document being parsed.

  These systems do not emphasize dynamism. Furthermore, join operations in profiles and overlap among user profiles have not been addressed by any of these systems.

- Matching profiles in content-based publish/subscribe systems
  Our work is not the first attempt to solve the problem of matching profiles in content-based publish/subscribe systems.

  In [FLPS00], an event notification service is described which embodies an event model, a subscription language and a set of matching algorithms. Events are a set of attribute-value pairs and subscriptions are conjunctions of constant predicates. A subscription is satisfied by an event if the bindings for the attributes provided by that event makes the subscription true. The matching algorithms provided

in [FLPS00] exploit predicate redundancy and dependencies between predicates. Some of them also distinguish between equality and non-equality predicates to improve efficiency. [ASS+99] presents a similar system where the subscriptions are pre-processed into a matching tree. Each node in this tree is a test on some of the attributes and the edges are results of such tests. Leaves of the tree contain the subscriptions. Matching is realized by traversing this tree to obtain the subscriptions at the leaves. Expected time complexity of the presented method is sub-linear in the number of subscriptions. Another important work that needs to be cited is [BCM+99]. It describes Gryphon system developed at IBM which aims at efficiently distributing large amount of data to many clients. In [BCM+99], particularly the problem of multicast where there are multiple brokers, each capable of event matching is presented. This kind of an architecture is to realize publish/subscribe in a distributed fashion. The brokers route events to other routers which are closer to the client that needs to receive the events. Hence, matching is not accomplished at a centralized location, but it is distributed among multiple brokers.

All of the above mentioned systems focused on constant predicates in subscriptions. We propose alternative algorithms for matching profiles that contain both constant and join predicates.

Additionally, as we shall discuss later, there are data structures designed to solve some problems in computational geometry which are closely related and applicable to our problem. Next, we present our solution to the profile matching problem. We defer the discussion of our contributions to Section 5.

3 Preliminaries

3.1 Basic Architecture

There are two major parties in a publish/subscribe system: data sources that are continuously generating new data and users that are continuously in need to receive new data. The functionality of the publish/subscribe system is to act as a medium for these two parties to reach each other. First, users submit subscriptions to the system. Later on, as new data arrives from the data sources, the system redirects the relevant ones to the subscribed users.

In a publish/subscribe system, users declare their interest in receiving data from the system through subscriptions. Subscriptions can take two major forms. In group-based publish/subscribe systems, the subscriptions simply specify the subject category the users wish to get subscribed. This means that they are willing to receive any document on that subject category. In content-based publish/subscribe systems, however, rather than getting subscribed for all the documents on a certain subject, users subscribe to some of the documents on that subject by declaring more detailed information on their data interests.

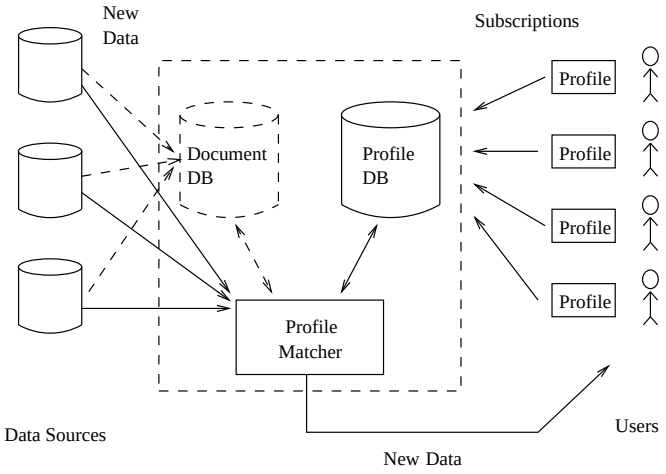


Figure 1: Publish/subscribe system architecture under consideration

In this project, we are considering a content-based publish/subscribe system where users get subscribed to the system through profiles (see Figure 1). Generally speaking, profiles are persistent queries which define the data that the users wish to receive. They both function as a tool that allows users to specify their data

interests as well as a filter that facilitates the delivery of data to interested users. The system stores these profiles in a Profile Database. On the other side, new data is arriving from the data sources. Although it is not a requirement, the system may also be storing the arriving data in a Document Database [1]. The crucial component in the system is the Profile Matcher which is responsible for deciding which users should receive a newly arrived data object. This decision has to be made based on the profiles stored in the Profile Database. The naive way of making the matching is by comparing each user profile against the data object one by one. However, this approach would not scale with the number of users (i.e. profiles) and also the rate of data arrival to the system. Therefore, our purpose is to find a more clever mechanism to perform the matching.

3.2 The Data Model and The Profile Language

Before we discuss how we can handle the profile matching problem, we need to state our assumptions about the data model and the profile language we are using.

We consider each data object/document as a collection of value assignments to attributes. It is similar to a record in the relational model. However, we do not assume that the record is in a predefined schema, i.e., there is no restriction on which attributes are allowed to appear in a document and what range and type of values they are allowed to take.

We take profiles each of which is a conjunction of constant and join predicates. It is like a simplified version of the `WHERE` clause of a relational query. However, it does not contain any negations or disjunctions. Predicates are simple equality or inequality constraints on the values that a set of attributes can take for the document to qualify for user's interest. Values are chosen from ordered domains like integers. Join predicates specify the constraints on attributes in a newly arrived document and an old document that may be stored in the Document Database. Old documents to be joined by the new document need to be quantified. Therefore we use either a universal or existential quantifier together with the join predicates. Universal quantifier ($\forall$) denotes that we want the relationship between the attributes to occur for all the documents that are stored in the Document Database and existential quantifier ($\exists$) denotes that we want the relationship to hold for at least one old document.

Below are some examples to illustrate profiles:

Example 1 *documents whose author is "Smith"*
`author = "Smith"`

Example 2 *call for papers for conferences in the area of "Databases" which are issued after 2000*
`area = "Databases" AND year > 2000 AND doc_type = "CFP"`

Example 3 *data that is about books by "Smith" whose price is in the range [10, 30] and which is different than the ones that have been received up to now*
`author = "Smith" AND 10 <= price <= 30 AND` $\forall$ `d (isbn` $\neq$ `d.isbn)`

4 Index Structures

We faced the problem of data matching in traditional database systems before. There, we had large amount of data stored in the database and that data needed to be matched against user queries by the query processor. To facilitate this matching, we built indices on the data. In publish/subscribe systems, the role of queries and data are reversed. Now we have large number of user queries (i.e. profiles) stored in the system and we wish to match a data object against these queries. Instead of indexing the data, now we can build indices on the queries to facilitate profile matching.

In this section, we first present what kind of indices can be used on constant predicates followed by indices we can use on join predicates. Then we describe how we can combine the two techniques. We also present performance evaluation for each of the index structures.

4.1 Index Structures on Constant Predicates

We developed two basic index structures: the Cluster Index and the Interval Index. We also tried mixing these two indices together to come up with a better index and we called this third index the Hybrid Index. It turned out that there are other data structures that were developed to solve a related problem in computational

[1] We call data objects *documents* since we think in terms of the web context.

geometry: the windowing queries problem [dBvKOS00]. Two of these data structures that can also be used for our problem are the Segment Tree and the Interval Tree [dBvKOS00]. We also implemented these two structures to compare with ours. In this section, we describe each of these five structures and present both theoretical and experimental comparison.

Before getting into details of each index, let us first mention about some basic principles that apply to these structures.

- We handle the problem one attribute at a time. Therefore, we build one index tree per attribute. To map each attribute to its corresponding index tree, we use hashing on attribute names.
- By handling each attribute separately, we reduced the problem to the following: Given a set of intervals for an attribute specified in the profiles and given a data point specified by the value assigned to the attribute in the new document, find the list of profile intervals which contain that data point.
- In all the structures, index tree is a variation of the binary search tree. At the end of the search performed on the tree, we end up with a set of profiles.
- The set of profiles are denoted by bit vectors. There is one bit per profile in the vector. A bit being 1 indicates positive view about the matching of the corresponding profile. In the Cluster Index, a bit vector shows which profiles *may* match whereas in the other indices, it shows which profiles *do* match.
- There are two methods used in building the index trees:
  - based on the endpoints of the intervals, as in the Cluster and the Interval Tree Index
  - based on the elementary intervals between consecutive endpoints, as in the Interval and the Segment Tree Index

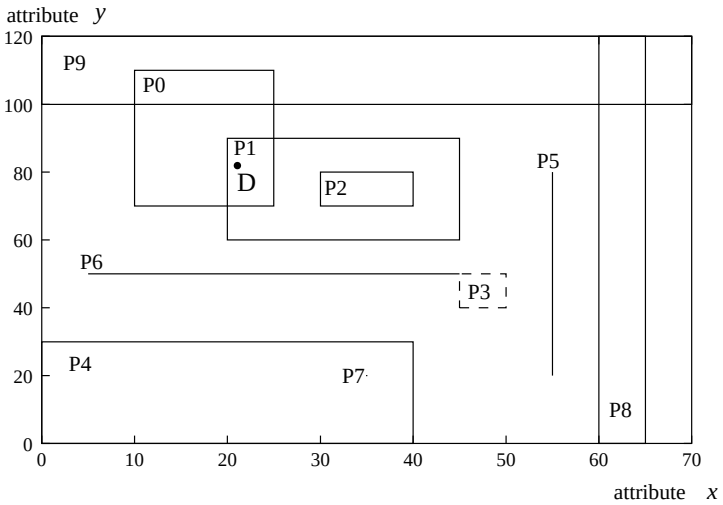
  The Hybrid Index involves both approaches.

We use the following example to illustrate each of the indices in the next coming sections:

Example 4

`Profiles:`
`P0:` $10 \leq x \leq 25 \wedge 70 \leq y \leq 110$
`P1:` $20 \leq x \leq 45 \wedge 60 \leq y \leq 90$
`P2:` $30 \leq x \leq 40 \wedge 70 \leq y \leq 80$
`P3:` $45 < x < 50 \wedge 40 < y < 50$
`P4:` $x \leq 40 \wedge y \leq 30$
`P5:` $x = 55 \wedge 20 \leq y \leq 80$
`P6:` $5 \leq x \leq 45 \wedge y = 50$
`P7:` $x = 35 \wedge y = 20$
`P8:` $60 \leq x \leq 65$
`P9:` $100 \leq y \leq 120$

`Document:`
`D:` $x = 20 \wedge y = 80$



4.1.1 Cluster Index

The main idea in the Cluster Index is to group profiles based on a given attribute's value. At each level of the index tree, the profiles are splitted into two groups. Tree node contains the value that is used as the splitting point, which we call the *key*. All the profiles for which the interval to be satisfied by the attribute lies to the left of the key (i.e. less than the key) go to the cluster represented by the left subtree and all the rest go to the cluster represented by the right subtree.

The keys are chosen from the set of endpoints of the intervals covered in the profiles for that particular attribute. First we sort the endpoints. Then we search for a good point to be chosen as the key. We use two criteria to decide how good a splitting point is:

- The tree should be kept as balanced as possible so that the depth of the tree does not become large.
- We should keep the number of profiles that appear in both subtrees (to the left of the chosen key and to the right of the chosen key) as small as possible. This is also to keep the tree size under control.

Since these two criteria are opposing with each other, we need to find a compromise. We use the following simple scoring function: (i) get the difference between the number of profiles going to the two subtrees to measure how balanced the split is, (ii) count the number of profiles that appear in both subsets to measure the amount of repetition, (iii) add these together. The key which results in the smallest score is chosen as the best key.

We perform the search for the key in a binary search fashion. We start by choosing the $\lceil n/2 \rceil$th point in the endpoint list and calculate its score. Then we do the same for the sublist which has more number of profiles. Each time we compare scores and keep track of the key whose score is the minimum. For example, assume that the sorted list of endpoints is $[2, 6, 19, 21, 23, 48, 50]$. First we take 21. Let's say 3 profiles lie to the left of 21 and 4 profiles lie to the right of 21 and the total number of profiles is 5. Then the score for 21 would be (4-3)+(4+3-5) = 3. Since the number of profiles to the right of 21 is more than the number of profiles to the left of 21, we choose 48 as the next candidate key. Let's say 5 profiles lie to the left of 48 and 2 profiles lie to the right of 48. Its score would be (5-2)+(5+2-5) = 5. Since $3 < 5$, 21 would be a better choice that 48. We continue like this until we examine the whole list (For this example, we only need to check 23 now).

In addition to the key, a bit vector is stored in each node which denotes the set of profiles that have the possibility of being satisfied (i.e. matched by the document) right before the split. This bit vector is computed depending on the chosen key at the parent node [2].

We continue splitting and computing the corresponding bit vectors until either we are left with clusters of size 1 or we can not split anymore. We decide that we can not split any more when whatever endpoint is chosen, one or both of the clusters are equivalent to the parent cluster. This occurs when the intervals have very high overlap. The nodes of the tree at the leaf level contain only bit vectors since no key is needed to split any further.

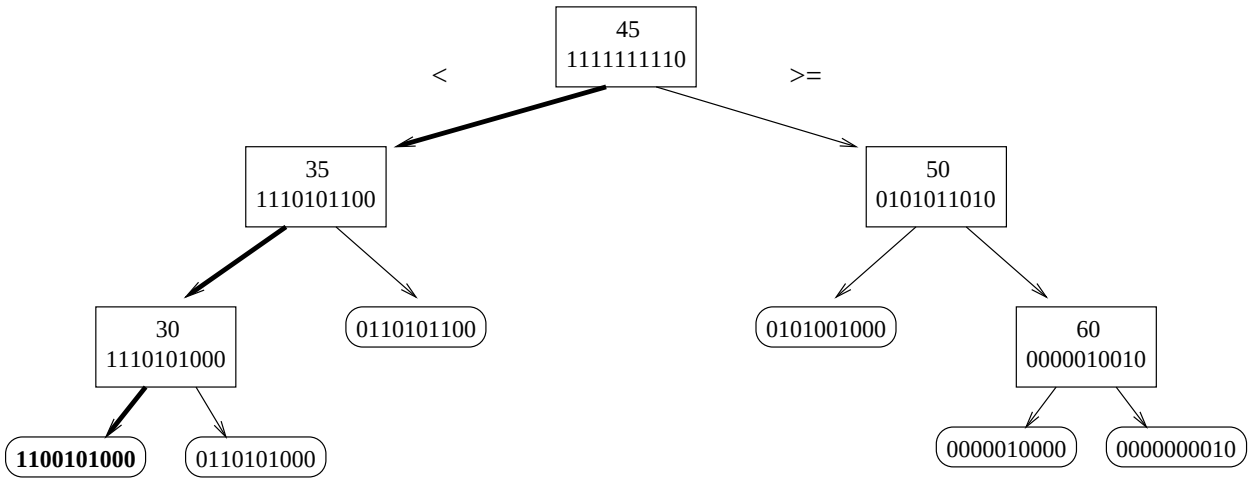


Figure 2: Cluster Index on attribute x

Figure 2 shows the Cluster Index tree for x attribute for the profiles given in Example 4. To match document D given in the example on this tree, we traverse the tree all the way down to the leaf level by making comparisons with the key at each node on the path. The bit vector at the leaf gives us the cluster of profiles which have the possibility of matching the document. The bold arrows in Figure 2 show the path to be followed to match x attribute of D which is assigned value 20. Note that we build a similar tree for the y attribute and we do a traversal to match y attribute of D. The bit vectors obtained after each traversal are propagated. After the final traversal, we end up with a cluster of profiles which should be checked to find the exact answer. The smaller the size of this final cluster is, the less number of profile evaluations are to be done at the end. Therefore, it is important that we have small clusters at the leaves of the index trees.

[2] For the Cluster Index, we actually do not need to store bit vectors in the inner nodes. We show them here for illustration.

4.1.2 Interval Index

Interval Index is a binary search tree built on elementary intervals. Elementary intervals are obtained from the partitioning of the range of values for a particular attribute created by the endpoints specified in the profiles. For example the list of elementary intervals created by the list of endpoints $[2, 5, 18, 30]$ is $[(-\infty, 2), [2, 2], (2, 5), [5, 5], (5, 18), [18, 18], (18, 30), [30, 30], (30, +\infty)]$. Hence, the list of elementary intervals consists of open intervals between two consecutive endpoints, alternated with closed intervals consisting of a single endpoint [dBvKOS00]. For n distinct endpoints, there are $2n+1$ elementary intervals. In this example, we show an infinite range for values but we can also limit the range using finite values. Note that each elementary interval is disjoint from all the others. Thus, if a given point is found to be in an elementary interval, it can not be in another elementary interval at the same time.

Each node in the index tree consists of an elementary interval and a bit vector. The bit vector denotes the profiles that would be satisfied if the attribute's value were found to be in that elementary interval. As we are building a binary search tree, we choose the $\lceil n/2 \rceil$th elementary interval from the sorted list of elementary intervals to place into the root node. Then we compute the corresponding bit vector. The chosen interval splits the list of elementary intervals into two equal size sublists. The sublist to the left is used to build the left subtree and the right sublist is used to build the right subtree in a similar fashion, until all the elementary intervals are placed into the tree. The smaller the number of elementary intervals, the smaller the tree size will be. Therefore, we apply the following optimization: Before we create a node for an elementary interval, we check whether there is any profile satisfied by that elementary interval. If no such profile exists, we do not need to create a node for that elementary interval.

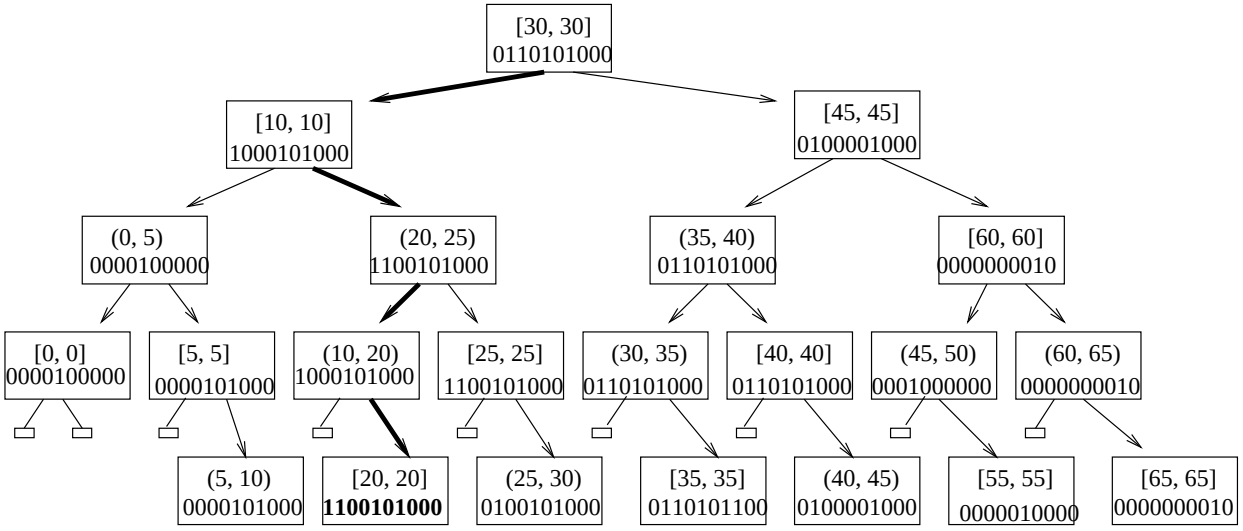


Figure 3: Interval Index on attribute x

Figure 3 shows the Interval Index tree for x attribute in Example 4. To match document D given in the example on this tree, starting from the root node, we search for the interval which contains the value 20 assigned to x. As soon as we find such an interval, we declare the bit vector stored in the corresponding node as the set of profiles that match as far as the x attribute is concerned. Hence, we do not always need to traverse until the leaf level. However, if we search until the leaf level and still can not come up with a matching interval, then we decide that none of the profiles match. The bold arrows in Figure 2 show the path to be followed to match the x attribute of the given example. As in the Cluster Index, we need to do a similar search for the y attribute as well and the resulting bit vectors are to be combined.

4.1.3 Hybrid Index

As we mentioned before, Cluster Index may not have good performance when the clusters at the leaves are of large size. The reason is that all the profiles in those clusters have to be evaluated one by one. This situation occurs when the overlap among profiles is high and we can not split them into smaller clusters.

We developed the Hybrid Index to solve this problem. The idea is to first build a Cluster Index and then

to build Interval Index on clusters at the leaves of the Cluster Index. This way we can avoid all profiles in the clusters from being evaluated as if no index existed on them.

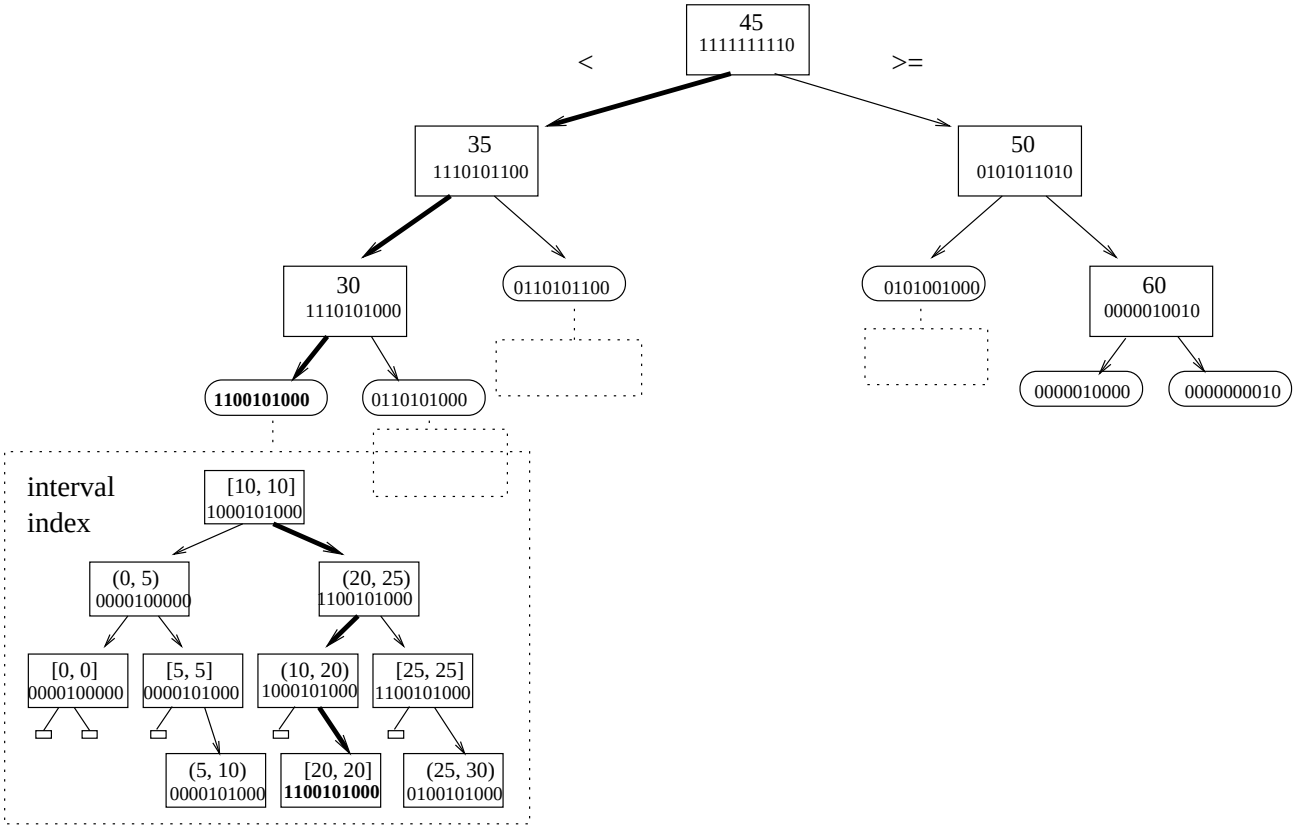


Figure 4: Hybrid Index on attribute x

Figure 4 shows the Hybrid Index tree for x attribute in Example 4. The top half of the tree is exactly same with the Cluster Index. Then we apply Interval Index to the leaf clusters of size larger than 1 [3]. We only illustrate one of the Interval Index trees in Figure 4 for convenience. To match document D, we perform the search as we do for the Cluster and the Interval Indices. Again, the bold lines show the path we follow to match 20 for x.

We expect the Hybrid Index to perform better than the Cluster Index when the degree of overlap between the profiles is high. We present its performance against both the Cluster and the Interval Indices in the following sections.

4.1.4 Geometric Data Structures

When we handle our problem of deciding which profiles match a given data object one attribute/dimension at a time, we end up with a problem that has been worked on before in the field of computational geometry: Given a set of intervals, report the ones that contain a given query point. Three known data structures developed to solve this problem are: Segment Tree, Interval Tree and Priority Search Tree. We exclude Priority Search Tree from our discussion since it is developed for a special case of windowing queries and we need to perform a transformation on the intervals and points to be able to use this structure [dBvKOS00]. Below we describe how we implemented and used the remaining two structures.

- **Segment Tree**
  Segment Tree structure is similar to our Interval Index in that it also is built on elementary intervals. However, elementary intervals are stored only at the leaves. Inner nodes store the union of their child

[3] Actually, it is not usually advantageous to apply the Hybrid Index to very small clusters. Therefore, we set a parameter *minimum cluster size* and start applying the Hybrid Index for clusters that are equal (or larger when no more splits are possible) to that parameter. To be able to apply this flexibility, we need to store bit vectors in every node of the Cluster Index.

intervals. The basic principle is to reduce the number of nodes where a specific interval is stored. If an interval is covered by a lot of elementary intervals, then it is better be placed higher in the tree. In our Interval Index, we do not try to minimize the number of times a profile interval is stored in the nodes because we are using bit vectors to store and process profile sets efficiently. However, we apply an optimization to reduce the number of elementary intervals to be stored, as mentioned before.

The way Segment Tree is built is also different. It is built in a bottom up fashion. Again we have an ordered set of elementary intervals. We combine them in pairs. Union of each pair of intervals is used to create a parent node. The root node is the union of all the elementary intervals.

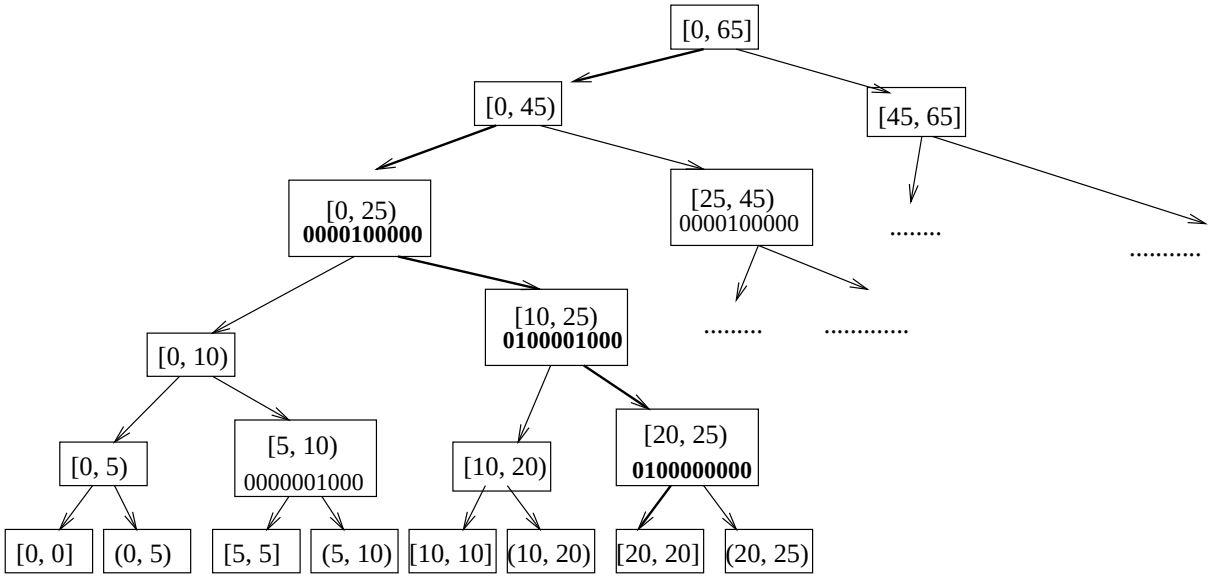


Figure 5: Segment Tree Index on attribute x

Figure 5 illustrates the index tree for the x attribute of our example. To match $x = 20$ on this tree, starting from the root, we check each interval until we reach a leaf node. The choice of whether we should follow the left branch or the right branch of the tree is made by checking the children intervals to see which of them contain the point. As we move from one node to one of its children, we collect the bit vectors. Taking a logical OR of all the bit vectors we collected gives us the list of profiles that match. In Figure 5, we do not show the whole tree and the bit vectors for all the nodes. We only show the part of the tree that is relevant to our example and the bit vectors that contain at least one 1's in them (because other bit vectors have no contribution to the resulting bit vector).

Note the important difference between the Segment Tree and our Interval Index in profile interval matching: in the Segment Tree, we must always traverse until we reach the leaf level to find the matching intervals whereas in the Interval Index, we stop traversing the tree as soon as we find a matching elementary interval. This difference is important in terms of the number of predicate evaluations (i.e. value comparisons) to be made.

- **Interval Tree**

This data structure, like the Cluster Index, makes use of the sorted endpoints rather than the elementary intervals. A chosen endpoint is again used to split the profiles into groups. However, no profile appears in more than one group. This is achieved as follows: when a point is chosen as the splitter, all the intervals that contain that point are stored at the node where that point is stored. Of the remaining intervals, the ones that are smaller than the point (i.e. to the left) are stored in the left subtree and the rest go to the right subtree. Additionally, the intervals to be stored at a node are stored twice: (i) as a list sorted in increasing order of their left endpoints, (ii) as a list sorted in decreasing order of their right endpoints. We did not use bit vectors for these because the order of each profile in the lists is important and this order can not be captured using bit vectors.

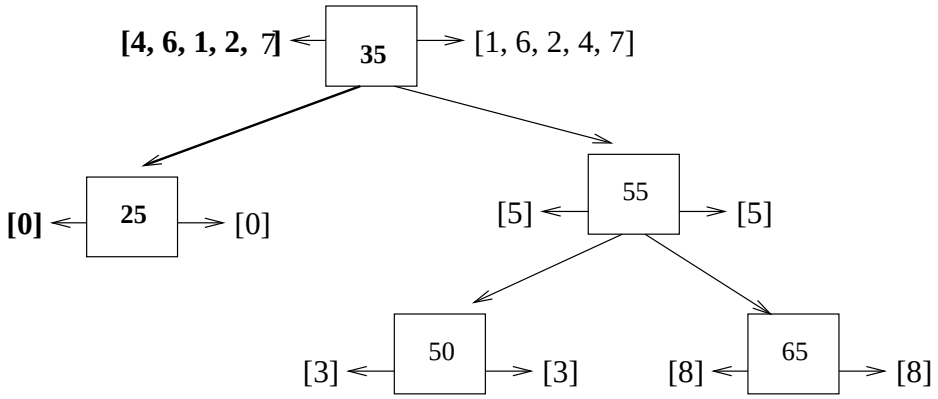


Figure 6: Interval Tree Index on attribute x

Figure 6 illustrates the data structure more clearly. To match $x = 20$, we do the following: Since $20 < 35$, we will proceed in the left direction. Before we move on to the left child node, we first need to check the list of profiles on the left in order. Starting at the interval with the leftmost endpoint, we report all the intervals that contain the point. As soon as we see an interval that does not contain 20, we stop and move on to the left subtree. In our example, intervals for profiles 4, 6, 1, 2 (given in bold) contain 20, but interval for profile 7 does not. Then we continue to the left. Since $20 < 25$, we need to again check the left list which only contains interval for profile 0. This interval also contains 20 and we report profiles $[4, 6, 1, 2, 0]$ as matching.

As illustrated, while profile matching, we not only traverse on the Interval Tree, but also search through the lists attached to the nodes. Therefore, even though the storage requirements seem to be advantageous for this type of tree, performance may not be good in terms of number of evaluations.

4.1.5 Theoretical Evaluation

We can analyze the performance of the data structures in terms of the following measures:

- **Build Time**
  This is a measure that shows how efficiently we can build the tree. This is not related to the matching performance of the indices that we are primarily after. However, it is still a useful measure. In all of the indices the primary determinant of this complexity is the sorting phase. Therefore, build time is $O(nlogn)$ per attribute, where n is the number of profiles. For k attributes, it becomes $O(knlogn)$.

- **Storage**
  We measure storage complexity in terms of the number of nodes. It is in the order of number of profiles for each attribute, i.e. $O(kn)$ in total. Note that at each node, we are storing bit vectors each of size n bits. Therefore, if we measure the space complexity in terms of bits, we need to multiply the complexity term by n. Although we have not implemented yet, we consider using compression techniques to store the bit vectors. Besides, we may not need to store the bit vectors at each node as they are if we consider parent-child relationships between the nodes and store only the bits that are not possible to generate from the bit vector of the parent/children.

- **Evaluation Time**
  Evaluation time corresponds to the number of predicate evaluations (value comparisons) to be made for matching. This is the most important measure for us since we are more concerned with the efficiency of the matching phase. Before we present the results, note that the naive algorithm which evaluates each profile in sequence without using any index would perform $O(kn)$. A more clever algorithm that could do better than the naive algorithm in absence of any index would be, what we call, the *lazy algorithm*. This algorithm would stop doing predicate evaluations in a profile as soon as it evaluates a predicate to false because profiles are conjunctions and a profile matches only if all of its predicates are evaluated to true. This algorithm would perform $\Omega(n)$ in the best case since it would have to evaluate only one

Index	Number of Evaluations
Cluster	$O(kn)$, $\Omega(klogn)$
Interval	$O(klogn)$, $\Omega(k)$
Hybrid	$\Theta(klogn)$
Segment Tree	$\Theta(klogn)$
Interval Tree	$O(kn)$, $\Omega(klogn)$

Table 1: Theoretical comparison of the indices in terms of number of predicate evaluations

predicate from each profile. The performance is still in the order of number of profiles and we expect to get better results from the indices.

Table 1 lists the results for n profiles with k different attributes. The two endpoint-based methods, the Cluster Index and the Interval Tree Index have the worst case performance of $O(kn)$. The reason for this is that, when the amount of overlap among profiles is really high, these indices would not be able to perform any splits. The index trees would be of depth 1 (only the root node) and we would have to evaluate each predicate in each profile one by one to perform the matching. On the other hand, when the overlap is not very high, then these indices would have attribute trees of depth $O(logn)$ and would achieve $O(klogn)$ number of comparisons. Whether the overlap is high or low, the Hybrid Index ensures that we do not do more than $O(klogn)$ number of predicate evaluations since it recovers the disadvantages of the Cluster Index by using Interval Index. Interval Index by itself has the worst case performance of $O(klogn)$, but can do better in the best case where the matching interval is found very early in the tree (at the root node for example). In this case, performance becomes independent of the number of profiles. Segment Tree Index has similar performance to the Interval Index except that its best case performance is equal to its worst case performance which is $O(klogn)$. Overall, our Interval Index has the best performance. It is also interesting to note that elementary interval-based methods, namely, the Segment Tree and the Interval Index (also the Hybrid Index indirectly) perform better and independently from the overlap among profiles.

4.1.6 Experimental Evaluation

We performed some simple experiments to verify our theoretical evaluations and to quantify the effect of overlap on performance which was not very obvious from the theoretical results. In our experiments, we concentrated on measuring how profile matching time in terms of the number of predicate evaluations changes with the increasing number of profiles. As we were expecting some changes in performance as the amount of overlap among profiles change for some of the indices, we also performed an experiment to analyze this change.

Figure 7 shows the results we obtained for our first experiment. In this experiment, we used 50 randomly generated documents. We also generated 100 to 1000 profiles. Both the documents and the profiles were containing 2 attributes of the same names. Predicates in the profiles were in the form of closed intervals of size 10. We used Zipfian distribution to control the generation of the profiles. Skew determined how dispersed the profiles would be from an origin. The direction of dispersion was determined uniformly at random. For this experiment, the overlap range was held low ([1, 20] percent) to exclude its effect. We would like to get a general idea of how the average matching performance of the indices scale with the increasing number of profiles. As the graph suggests, all indices perform well compared to no-index case where we would have to do $2 * 100 = 200$ to $2 * 1000 = 2000$ predicate evaluations on the average for 50 documents. This number changes in [18, 47] for the worst performing index in Figure 7. Although all performed well, we can see that our Interval Index and Cluster Index were the best of all. It was interesting to see that Hybrid Index did not beat Cluster Index at low degrees of overlap. We explain this result as Hybrid Index paying a constant penalty when it switches from one type of index to the other.

The first experiment does not tell us how overlap among profiles affect the performances. In our second experiment, we took 100 profiles and 50 documents generated in the same way. This time instead of varying the number of profiles, we varied the size of the intervals covered by the profiles to be able to change the overlap among them. We obtained overlap percentage in the range of approximately [10, 98]. The result is provided in Figure 8. It shows that the Interval Index and the Segment Tree Index are not influenced by the overlap increase and they have a steady performance. Both the Cluster Index and the Interval Tree Index are

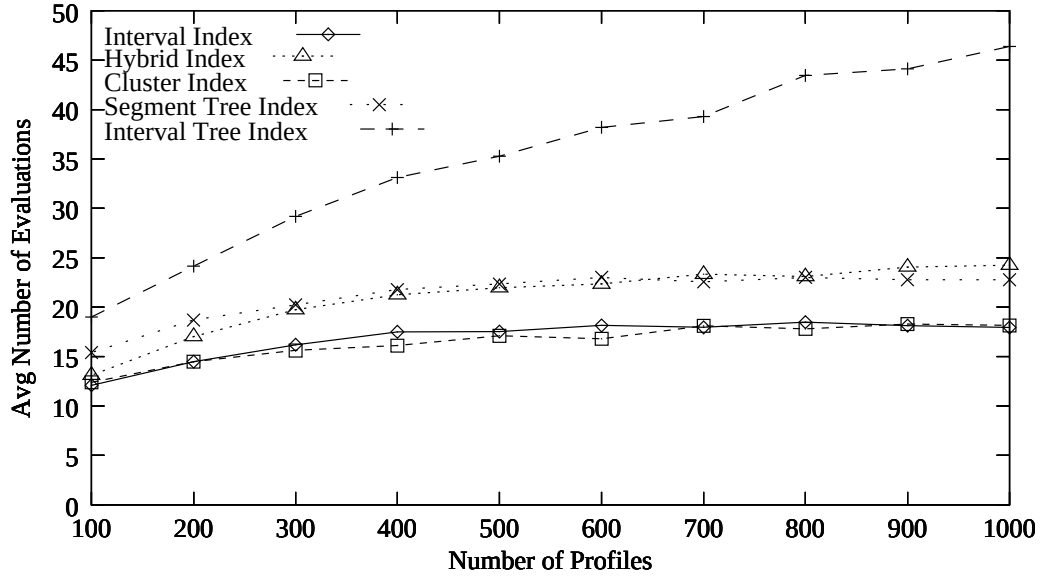


Figure 7: Average number of evaluations over 50 documents vs Number of profiles

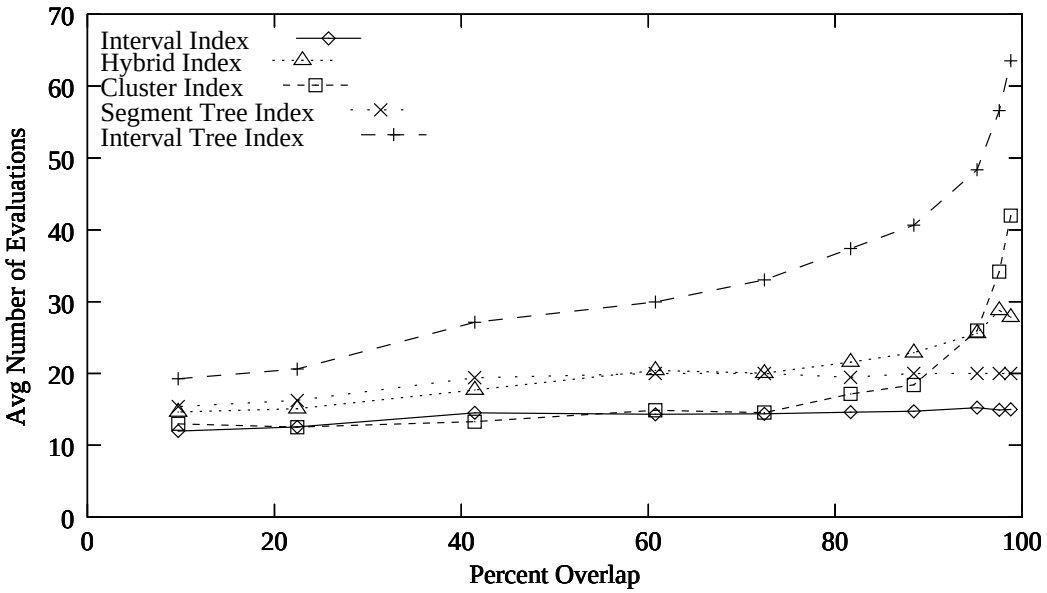


Figure 8: Average number of evaluations over 50 documents vs Percent overlap

highly affected from the increase in overlap. Hybrid Index takes over the Cluster Index as we expected but it can only do this after a very high degree of overlap (above 90 percent).

As a result, our Interval Index performs always better than the others. Our Cluster Index performs well as long as the overlap is not very high.

4.2 Index Structures on Join Predicates

A user's data interest may be spanning several documents. An instance of this situation is where the user wishes to correlate between some document that he has received before (or even if he has not received, he imagines that it is probable that the system might have received it/some source might have generated it) and a new document. For example, the user might be interested in documents published after all the documents he has received before (or all the documents being stored in the Document Database). This requires a comparison

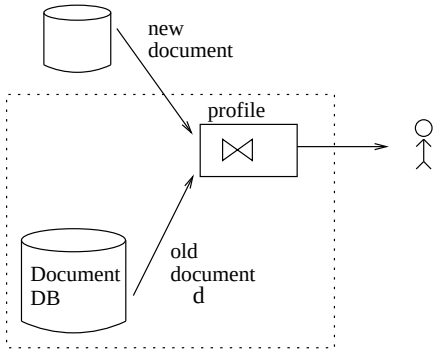


Figure 9: Joins between old documents and a newly arriving document

between the dates of all the old documents and the date of the newly arriving document. There may also be cases where the user wants this comparison to hold at least for one old document.

As seen in Figure 9, this comparison is similar to performing a join operation between two relations in a relational database. The major difference is that now one of the relations contain only one record (the new document). Also, now we have the queries (i.e. profiles) and one of the relations (i.e., old documents) stored in the system, but we do not know the other relation (i.e., new document) until it arrives to the system.

To accomplish profile matching efficiently when we have join predicates, this time we need to index the join predicates. However, we also have old documents participating in the joins and they also need to be efficiently accessed. Therefore, together with the join predicates, we need to index the data stored in the Document Database.

The documents in the Document Database can be indexed using conventional index structures like B-Trees [Bay72]. However, we consider augmenting such an index structure with document bit vectors (similar to profile bit vectors used in the previous section) for our purposes. These bit vectors will be used later to determine which old documents are likely to satisfy the join predicates given in the profiles. Each such vector will be of length equal to the number of documents in the Document Database and a bit of value 1 will indicate that the corresponding document is satisfying the join predicate. As in constant-only predicates, we consider that we have one augmented B-tree per attribute. We are not getting into details of this index here, but we will show how it is used in the following paragraphs.

To match join predicates efficiently when a new document comes to the system, we use a join table. Each entry of this table corresponds to either a simple join expression consisting of one quantified join predicate or a complex join expression which is basically a pointer to a quantified expression tree which has AND operators in the inner nodes and simple join predicates at the leaves. We do not store the complete tree in the entry but we just point to simple join predicates that are to be found on the leaves which are stored as separate entries in this table.

Example 5 better illustrates the index we are proposing and how it is used for matching.

Example 5

Profile Database:

P1: ∃ d (price < d.price)
P2: ∀ d (year > d.year)
P3: ∃ d (author = d.author)

Document Database:

D1: price = 10 AND year = 2000 AND author = "Smith"
D2: price = 25 AND year = 1999

Join Table:

join expressions			*document bit vector*	*profile bit vector*
quantifier	*join pair <old, new>*	*operator*		
∃	<price, price>	<	**00**	*100* ×
∀	<year, year>	>	**11**	*010* √
∃	<author, author>	=	**10**	*001* √
				011

New Document:

price = 30 AND year = 2001 AND author = "Smith"

In Example 5, three profiles are stored in the Profile Database and two old documents are stored in the Document Database. All of the profiles only contain simple quantified join predicates. We store these predicates in a join table. For each entry in this table, we also store a document bit vector and a profile bit vector. The document bit vector column is initially empty. We can only fill it out when we receive the new document. However, the vectors in the profile bit vector column can be computed at compile-time. A profile bit vector shows which profiles would be satisfied if the corresponding join predicate were evaluated to be true. Again, there is one bit per profile and a bit value of 1 indicates that the corresponding profile would be satisfied.

At run-time, i.e., when a new document is received at the system, we apply the following procedure for profile matching on the join table:

```
current profile bitvector = 000
for each entry of the Join Table with new join attribute appearing in the new document
    get the document bit vector from the B-tree on the relevant attribute
    if existential quantifier and there is at least one bit 1 in the document bit vector
       current profile bitvector = OR(current profile bitvector, profile bit vector)
    else if universal quantifier and all bits in document bit vector are 1 then
       current profile bitvector = OR(current profile bitvector, profile bit vector)
output current profile bitvector
```

How we applied this procedure to the example is also illustrated above. Boldface bit vectors show the bit vectors computed at run-time. Since the first join predicate is quantified as existential and there is no 1 bit in the corresponding document bit vector, we do not use the first profile bit vector. On the other hand, for the second join predicate, all the bits in the document bit vector are found out to be 1 and since this predicate is universally quantified, we take the corresponding profile bit vector (`010`) and `OR` it with the initial profile bit vector (`000`), obtaining `010`. The third entry's document bit vector contains one 1 bit and since the predicate is existentially quantified, its corresponding profile bit vector (`001`) should also be taken and `OR`ed with the current profile bit vector (`010`), giving us the result `011`. This result says that given the Profile Database and the Document Database, when a new document as given in the example arrives, the matching profiles are `P2` and `P3`.

Organizing join predicates this way has the following benefits:

- Each distinct join predicate is considered at most once.
- Augmented B-Tree enables us to find the matching old documents easily in logarithmic time.

4.3 Using the Index Structures on Constant and Join Predicates Together

Until now, we have presented how we handle constant and join predicates separately. In this section, we discuss how we can use them in combination.

4.3.1 How to Apply them Together

Each index structure basically functions as a filter against profiles. Each time we complete using an index, we are left with the collection of profiles that still have the possibility of matching the given data item. Hence, we consider application of constant and join index structures as two major phases in this filtering process: (i) we can first apply one of them on the complete set of profiles, (ii) then we apply the other one on the set of profiles that survive the first phase. In other words, the resulting profiles obtained by passing profiles through one of these indices in the first phase will be further filtered through the other index in the second phase.

Filtering based on join predicates is supposed to have nearly the same cost with filtering based on constant predicates. The reason is that we are building B-tree-like indices on each attribute in the document base that is being joined with an attribute from the new document as stated in the profiles. Therefore, both the join index and the constant index require logarithmic time per attribute. The difference will come from the difference between the number of different attributes in constant predicates and the join predicates and from the difference between the number of old documents (D) and number of profiles (N).

Here is a rough comparison of different cases:

1. when there is no index at all:

```
for each profile
    for each document
        if document matches the profile
           report the profile
           break;
```

N: number of profiles
D: number of documents
c: average number of conjuncts per profile

Worst case number of predicate evaluations = $N * D * c$

2. first phase: filter through the index on join predicates
   second phase: filter through the index on constant predicates

   j: number of join predicates/conjunctions
   k': number of different attributes in constant predicates which also appear in the new document after the first phase is applied

   Worst case number of predicate evaluations = $j * logD + k' * logN$

3. first phase: filter through the index on constant predicates
   second phase: filter through the index on join predicates

   k: number of different attributes in constant predicates which also appear in the new document ($k \geq k'$)
   j': number of join predicates after the first phase is applied ($j' \leq j$)

   Worst case number of predicate evaluations = $k * logN + j' * logD$

It is out of question that having an index is more efficient than not having any. Choice between 2 and 3 highly depends on the relative values of D, N and j, j' and k, k'. We can not know k' and j' before we see a new document. However, given j, k, N and D (and maybe some statistics about the distribution of coming documents), we can estimate which phase is more beneficial to apply first.

4.3.2 Pre-filtering

Constant predicates impose certain constraints to be satisfied by the newly arriving documents whereas join predicates impose constraints to be satisfied by the documents in the Document Base as well as the new documents. Filtering based on each of these group of constraints in isolation is only possible after a new document arrives to the system. However, when we consider the two groups together, we may have an opportunity to perform some filtering at compile-time (i.e., before the new document arrives) based only on the old documents. We call this early filtering process *pre-filtering*. Pre-filtering may provide efficiency in run-time filtering by reducing the number of profiles to be matched and hence preventing the use of certain parts of the indices we have built.

The above mentioned case is not the only case that provides opportunity for pre-filtering although it is the most important one. Below we list the cases in which we can apply pre-filtering:

- Existence of a given attribute:
  For each `d.attribute` that is found in each profile, the Document Database should at least contain one document `d` which has an attribute named `attribute`. In other words, a B-tree should exist on attribute named `attribute`. Otherwise, that document base can not satisfy the corresponding profiles and need not be checked when a new document comes.

- Constant predicates on old documents:
  For each `d.attribute OP constant` or `constant1 OP1 d.attribute OP2 constant2` in each profile, this predicate should be satisfied by at least by one document in the Document Database. If a universal quantifier is also provided, then the predicate should be satisfied by all the old documents.

- Constant predicates and join predicates together:

  Example 6
  `price = 10 AND ∃ d (d.price = price)` *implies the constraint* `∃ d (d.price = 10)`

  As Example 6 illustrates, we can infer new and compile-time-applicable constraints when we consider constant and join predicates together. In this particular example, the inferred constraint may replace the join predicate. However, this may not always be the case. Example 7 below shows a case where we would lose information if replaced the join predicate. Rather, we should use the inferred constraint as an additional one which still can contribute to pre-filtering.

  Example 7
  `price < 10 AND ∃ d (d.price = price)` *implies the constraint* `∃ d (d.price < 10)`

  We need to generalize the exemplified cases to be applicable to other similar cases. For each `(d.a OP1 a) AND (a OP2 c)` in each profile where `a` is an attribute name, `c` is a constant and `d` refers to an old document in the Document Database, Table 2 can be used to reveal implicit constraints on document `d`: [4]

OP1 / OP2	=	<	≤	>	≥
=	d.a = c	d.a < c	d.a ≤ c	d.a > c	d.a ≥ c
<	d.a < c	d.a < c	d.a < c		
≤	d.a ≤ c	d.a < c	d.a ≤ c		
>	d.a > c			d.a > c	d.a > c
≥	d.a ≥ c			d.a > c	d.a ≥ c

Table 2: Generalization of implicit constraints to be inferred from `(d.a OP1 a) AND (a OP2 c)`.

  The first row in the Table 2 (where `OP2` is =), contains the constraints that can replace the join predicates. The rest should be used as additional constraints. We do not show it here, but a similar table could be built for constant predicates in interval format, i.e., `constant1 OP1 attr OP2 constant2`.

To above cases should be identified and pre-filtering should be applied in an efficient fashion. Briefly, our approach to this problem is to list all the implied constraints for each profile; then to treat each of these the same way we treat the profiles themselves and build an index similar to our indices for constant predicates; and finally to pass each old document in the Document Database through this index and obtain a *pre-filter bit vector* which shows the profiles that survived. As new documents get added to the Document Database, we need to pass them through the pre-filter first and update the pre-filter bit vector accordingly.

Once we have a pre-filter bit vector, we use it as our starting set of profiles to apply the profile indices.

5 Contributions

We believe our work has the following contributions:

- We proposed three new index data structures to facilitate matching of constant predicates. We showed that one of our index structures, Interval Index, performed slightly better than two existing data structures in terms of best case performance. In terms of worst case, its performance is similar to the Segment Tree. All of the structures under investigation perform far better than the naive and lazy algorithms in general. We discovered that some of the indices show sensitivity to the amount of overlap among profiles. We conducted experiments to analyze this.
- Our profiles allow joins between the set of previously received data and a newly arriving data. To our knowledge, there is no previous work on matching join-capable profiles efficiently.

[4] Attribute names in join predicate do not necessarily have to be the same. However, it is important that the attribute name referring to the new document in the join predicate be the same as the attribute name in the constant predicate.

- We used bit vectors both to represent the set of matching documents as well as the set of matching profiles. These bit vectors enable us to perform set operations like intersection and union easily, simply by taking ANDs and ORs of the vectors, respectively.

6 Future Work and Conclusion

We would like to investigate the following problems in the future:

- Profile indexing and data delivery for disjunctive profiles with utilities
  Right now, the user can specify one data item of interest since we allow only conjunctive profiles. When we allow disjunctions in the profiles, he can also specify multiple data items that he wishes to receive. We can also let him specify preferences among these items in terms of utility or priority. Then we need to match and deliver the higher utility items earlier than the other ones. Moreover, since our focus was on the efficiency of the profile matching, we have not considered how the delivery of matching data items to multiple users can be performed efficiently. This problem also requires further investigation.

- Modifications to the index structures when the profiles are updated
  Right now the index structures we proposed are all static. Modifications would require them to be built from scratch. It is not very likely that users would want to update their profiles very frequently since they usually specify long-term data interests. However, it is expected to be quite often that new users get subscribed to the system. Therefore, we should develop methods for easy insertion of new profiles to the existing index structures. Other kinds of modifications (like deleting users, updating existing profiles) should also be handled afterwards.

- Efficient storage of indices on disk
  We assumed that the indices fit into the memory and have not considered how they could be mapped to disk so that they can be efficiently retrieved later. This is also an important issue to consider as future work. Also, we need to investigate the compression techniques to store the bit vectors efficiently as well.

To sum up, in this work, we investigated how we can realize efficient profile matching using indices on profiles. Our results show that evaluation time scales well with the increasing number of profiles when indices are used. Furthermore, we showed that the overlap among profiles can affect the performance of some of the indices and can perhaps be exploited to build better indices.

References

[AF00] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *International Conference on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 2000.

[ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. A., and T. D. Chandra. Matching Events in a Content-based Subscription System. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–61, Atlanta, GA, May 1999.

[Bay72] R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[BCM+99] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 262–272, Austin, TX, June 1999.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[FLPS00] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient Matching for Content-based Publish/Subscribe Systems. Technical report, INRIA, 2000.

[HCKW90] E. N. Hanson, M. Chaabouni, C. Kim, and Y. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 271–280, Atlantic City, NJ, June 1990.

[HJ96] E. N. Hanson and Theodore Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip Lists. *Information Systems*, 21(3):269–298, May 1996.

[OM97] L. Obermeyer and D. P. Miranker. Evaluating Triggers Using Decision Trees. In *International Conference on Information and Knowledge Management (CIKM)*, pages 144–150, Las Vegas, NV, November 1997.

[YGM94] T. W. Yan and H. Garcia-Molina. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM Transactions on Database Systems (TODS)*, 19(2):332–364, June 1994.