Implementing block-stored prefix trees in XML-DBMS

© Oleg Borisenko Ilya Taranov

Institute for System Programming, Russian Academy of Sciences
borisenko@ispras.ru, taranov@ispras.ru

Abstract

The problem of search efficiency through large amount of text data is well-known problem in computer science. We would like to introduce a *BST* data structure that allows searches through a set of string values, and is optimized for reading and writing large blocks of data. This paper describes the algorithms for insertion, deletion and search of variable-length strings in disk-resident trie structures. This data structure is used for value indexes on XML data. We also compare our implementation with existing *B+ tree* implementation and show that our structure occupies several times less space with the same search efficiency.

1 Introduction

The problem of developing data structures that provide basic dictionary operations (insert, deletion and lookup) and is optimized for disk storage has been investigated in computer science for a long time but remains very important. This work describes a new disk-resident data structure *BST* that stores a set of string keys and algorithms for insertion, deletion and search for this structure. Also we compare the existing implementation of *B+ tree* with our novel approach within Sedna XML DBMS [8, 18] and show that the latter has an opportunity for significant economy of disk space providing the same search time as *B+ tree*.

Our structure has been developed for use as a value index backend in Sedna XML DBMS. Sedna XML DMBS has native support for value indices on nodes and it has the following requirements to the structures used for index management:

1. Structure must provide the ability to store keys of unlimited length. For example, URI's can be very large in practice.

2. Structure must implement the concept of multimap abstract data type. An index for an XML document in Sedna may have duplicate (key, value) pairs.

2 Review of existing solutions

Most common data structures for implementaing database index are *B-tree*[3] and its variants[4, 19, 13]. *B-trees* are widely used in modern databases for indexing [6]. Storing the keys of arbitrary length in *B-trees* is possible however causes implementation problems. It is difficult to implement the efficient storage of variable length keys. In practice usually only a limited part of the key is stored in the tree node, and the remainder is stored in separate overflow pages[2, 16]. This approach is quite effective if the keys are short or differ only in first characters.

The other problem arises if one key corresponds to a set of different value nodes; it is difficult to implement an efficient search by key/value pair. Most of *B-tree* implementations do not provide an ability to store such multimaps. At the same time our indexes may have duplicate key/value pairs. To be able to store identical logical[1] keys the concatenation of key/value pairs is usually used as a physical key.

This article describes an index implementation for disk-resident usage that fits our requirements. It can be shown as an extension of a data structure known as a *trie*[5] which is also called *prefix tree*.

The idea of using *prefix trees* as a replacement for *B-trees* has already been discussed in other works. A modified B-tree, called S(b)-tree [7], stores a binary *Patricia*[17] tree in its nodes. The feature of *S(b)-tree* is that nodes do not store a key itself, but the number of bits passed during the comparison. During the search for the string you may have to compare this string with the string stored in external memory. However, such a comparison is not a big problem in itself. The problem is to store all string keys in a separate location. *S(b)-tree* is presented as a data structure to support full-text index and it is good enough for this task[12].

The *B-trie*[1] is very similar to our approach. The basis of this work was the implementation of a prefix tree that provides efficient usage of CPU cache [9]. Both structures propose effective partition of the prefix tree into blocks (buckets). But in our work we do not use stored supplementary data structures and provide more effective strategy to keep the blocks filled.

In addition, there are quite simple implementations of similar data structures, i.e. *Index Fabric*[14], which is a B-tree with keys stored in a prefix tree in internal nodes.

3 Building the structure

In this paper we introduce a *prefix tree* based data structure which is optimized for reading and writing large

[1] Here the *physical key* is the key that is actually used in comparison and search. *Logical key* refers to a key that is passed to index system interface.

Proceedings of the Spring Researcher's Colloquium on Database and Information Systems, Moscow, Russia, 2012

blocks of data. We call our structure the *Block String Trie* or *BST*. Let the set of stored strings be denoted by $K = \{s_1, s_2, \ldots, s_n\}$. The basic operations for our structure are defined as follows:

1. Insert string s to the set K.
2. Find all the strings with the prefix s in the set K.
3. Remove string s from the set K.

Such structure implements a set abstract data type but not a map. Now we specify how exactly we store the (key,value) pairs in our data structure. Each pair can be represented as $s = k + c + v$ (we call it *physical key*), where k is a key (or *logical key*), v is string representation of a value, and c is a symbol that is absent in the alphabet that k is built of [2]. To find all (key,value) pairs, that correspond to a given key k we look for all strings prefixed with $k + c$. Thus the dictionary problem can be solved using the introduced data structure.

3.1 Prefix tree

Our data structure is a rooted tree T which stores a set K of strings. The structure is as follows:

1. Each vertex x of tree T has the following properties:
   (a) A prefix $prefix(x)$ is a string which may have zero length.
   (b) An array $E(x)$ of $n(x)$ exiting edges ordered in lexicographic order of characters $c(e)$ they are labeled with.
   (c) Auxiliary flags (will be described as they appear further in text).
2. Each edge $e = (x, L_i(x))$. $L_i(x)$ is a node, pointed by i-th edge from the array $E(x)$. Each edge is labeled with character $c(e)$. In this case we consider a character the string of the length of one.
3. Any path $S(x_1, x_n) = x_1 e_1 x_2 e_2 \ldots e_{n-1} x_n$ in the tree is string s, obtained from the path $S(x_1, x_n)$ by concatenating its substrings $s = prefix(x_1) + c(e_1) + prefix(x_2) + c(e_2) + \cdots + c(e_{n-1}) + prefix(x_n)$. We also introduce the flag $final(x)$, that shows whether a node has a corresponding key from the set K. Thus the string s defined by $S(x_1, x_n)$ belongs to the set of strings K if and only if the vertex x_n is labeled with flag $final(x_n)$.

The last property implies that a set K may have more than one corresponding tree T. To build a tree T' that represents the same set as tree T we can add vertex x' with an arbitrary prefix and not marked as $final(x')$. Both trees T and T' will represent the same set of strings K. Thus it seems reasonable to introduce an additional condition, which covers such situations:

4. Each vertex $x \in T$, is such that $n(x) \leq 1$, is marked as $final(x)$. A tree that satisfies that condition is called *minimal*. Each vertex x that satisfies (doesn't satisfy) the condition $n(x) \leq 1 \Rightarrow final(x)$ is called *important (redundant)*.

[2] For text strings, we use null character.

If this condition holds, the set K uniquely defines a tree T.

Theorem. *Any non-empty set of strings K defines one tree T and the one tree only for which the conditions 1–4 holds.*

Proof. Let us assume that this statement does not hold and the set of strings K does not define the single tree T. This may happen in two cases: the set of strings K does not define any tree at all, or the set K defines more than one tree. We are not going to consider the first case, as any non-empty set K defines at least one tree construction procedure.

Let us assume that the set of strings K defines more than one tree T that satisfies the conditions 1–4. Consider two trees which corresponds to a set K. That means that there is a string $k \in K$ that has paths $S(x_1^1, x_n^1)$ and $S(x_1^2, x_m^2)$ in trees T and T' respectively, such as:

$$\begin{aligned} k &= prefix(x_1^1) + c(e_1^1) + prefix(x_2^1) + \\ &+ c(e_2^1) + \cdots + c(e_{n-1}^1) + prefix(x_n^1) = \\ &= prefix(x_1^2) + c(e_1^2) + prefix(x_2^2) + \\ &+ c(e_2^2) + \cdots + c(e_{m-1}^2) + prefix(x_m^2) \end{aligned}$$

The fact that trees differ means either of the following:

1. In trees T and T', there are vertices x_i^1 and x_i^2, such that $prefix(x_i^1) \neq prefix(x_i^2)$.
2. In trees T and T', there are edges e_i^1 and e_i^2 such that $c(e_i^1) \neq c(e_i^2)$.
3. In trees T and T', there are vertices x_i^1 and x_i^2, such that one vertex has the flag $final(x_i^{1|2})$, while the other does not.

First two cases contradict the definition of our tree and the third case is in conflict with the condition 4. That means that trees T and T' are equal, hence there is only one tree, corresponding to the set K. □

We consider only minimal trees in further sections unless stated otherwise. However the condition (4) is not required in the implementation of the structure and may be not held if *lazy removal* technique is implemented.

3.2 Block separation

The structure described in the previous section is designed primarily for storage and retrieval of strings in memory. But we require the structure, that can be stored in external memory and our algorithms should effectively operate with reading and writing blocks of the fixed size.

First of all we introduce a special type of vertex that does not store the prefix and links to other vertices and keeps only a reference to a node located in another block. We call such vertices *reference nodes* and we mark them with the service flag $external(x)$. It should be noted that the nodes that are referenced by a reference vertices, can not be reference nodes themselves.

In addition we introduce the notion of *twig*.

A *twig* is a rooted tree B such that all leaf vertices are marked as either $final(x)$ or $external(x)$. If a tree T has no reference nodes it consists of a single twig.

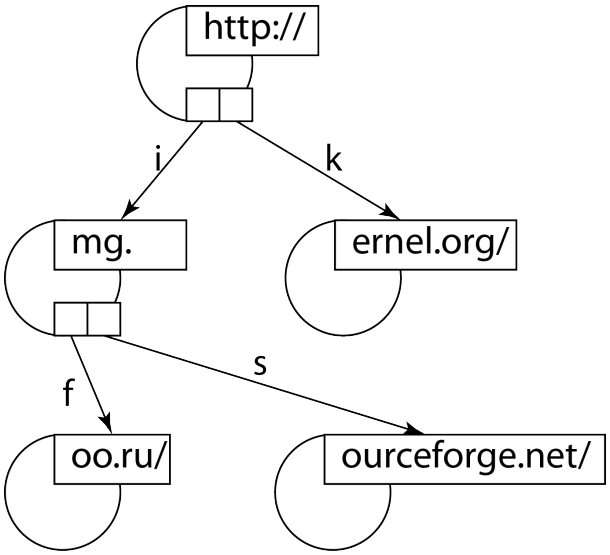

Figure 1: BST example without blocks

The concept of twig is a key for the problem of dividing BST into blocks. Every tree T can be divided into twigs by inserting a new reference node before any other node. Twigs

Blocks consist of the constant amount bytes W[3]. The distribution of twigs into blocks is organized in such a way that in one block one or more twigs from the initial tree can be stored. All the twigs in one block have a common direct ancestor. These conditions are necessary to ensure that the locality of changes in the tree and for block level locking. These conditions also imply that the block containing the root node has no other twigs.

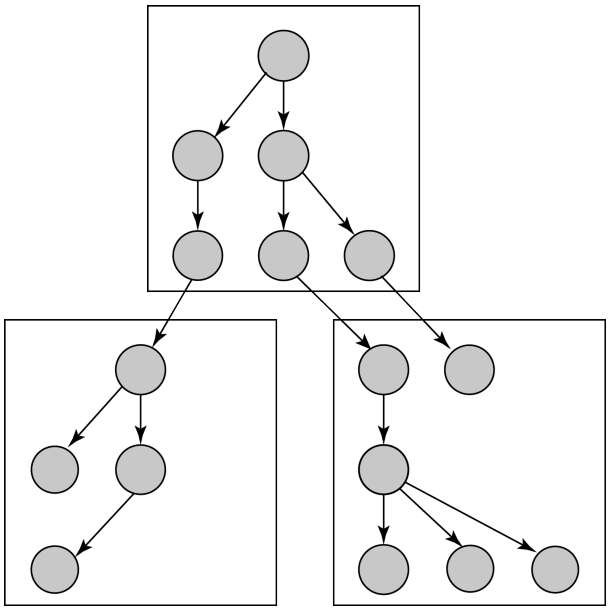

Figure 2: BST block separation example

[3]In the Sedna DBMS the default block size is 64KB.

3.3 Algorithms

3.3.1 Search

The search algorithm for the BST requires two input parameters: a pointer to r to the root node of the tree and string k to be found. The algorithm $BST-Search(r, k)$ returns a node x_{n+1} which satisfies the following:

1. String $s(x_n)$ defined by path $S(r, x_n)$ is a prefix of the string k or equals it if exists.

2. The search string k is a prefix of a string $s(x_{n+1})$ or equals it.

We define the relationship $A \leq B$ as follows: string A is a prefix of B or equals it. Thus our requirements for result may be expressed using the following inequality: $s(x_n) \leq k \leq s(x_{n+1})$.

The search procedure starts from the root of the tree. The function takes a pointer x to the root node and the search string k, an intermediate result is stored in the stack S. In addition there is also an auxiliary function $y = L(x, c)$ which returns node y which is pointed by that edge of vertex x, which is marked with character c (or NULL if there is no such an edge). This feature is implemented using binary search algorithm and the edges are stored in sorted array in our implementation. We also introduce the function $Cut(p, s)$ which returns a string, that is produced by removing the prefix p from a string s, so that $p + Cut(p, s) = s$. The search algorithm can be written in compact form as follows:

BST-Search(x, k, S)
1: Push(S, x)
2: **if** $external(x)$ **then**
3: Disk-Read($J(x)$)
4: **return** BST-Search($J(x)$, k, S)
5: **end if**
6: **if** not Is-Prefix($prefix(x)$, k)) **then**
7: **if** Is-Prefix(k, $prefix(x)$) **then**
8: **return** x
9: **else**
10: **return** NIL
11: **end if**
12: **else**
13: $s \leftarrow$ Cut($prefix(x)$, k)
14: **if** Empty(s) **then**
15: **return** x
16: **else if** $L(x, s[1]) =$ NIL **then**
17: **return** NIL
18: **else**
19: **return** BST-Search($L(x, s[1])$, Cut($s[1]$, s), S)
20: **end if**
21: **end if**

The search function returns node x such that all the paths from the root to all possible leaf vertices which contain node x define the strings prefixed by a search string. The function returns $NULL$ if there is no such a node. We can find all strings prefixed by k traversing the tree starting from node x.

3.3.2 Insertion

The insert operation begins with building the structure, that describes the modifications we are going to make in our tree. If some block on the way of our modifications does not enough space for modifications to be made, the block is split (this process is discussed in the next subsection) and the insertion process is invoked again.

The insert procedure starts with executing the $BST - Search(k, r, S)$ (where k is the string to be inserted) method, but we need only the path S as result. Also we will need three additional strings derived from path S and string k: $common$, $rest$ and key. We construct these strings in the following way. First of all, we need the string s', that is the string that corresponds to S with the last vertex's prefix removed. From the definition of search procedure it is obvious that this string is a prefix of added string k or equals it. We also need the string k' which is built from k by removing s' from the beginning. To build the string common $common$ we take the greatest common prefix of strings k' and $p = prefix(S[n])$, where $S[n]$ is the last vertex of the path S. In turn the strings $rest$ and key are p and k' with no prefix $common$ at the beginning respectively. The idea becomes clearer if illustrated by the following scheme:

We consider five cases in our insertion procedure:

1. There are no nodes in the tree. In this case, we need to allocate new page for a new node.

2. $Rest$ and key are empty strings. In this case, it is sufficient to mark the latest node of the path as $final$ (in case of a minimal tree it will already be marked as $final$).

3. Key is an empty string, $rest$ is non-empty. In this case, we should split the final node of the path in two parts. One of these parts contains the prefix $common$ and is marked as $final$.

4. Key is a non-empty string, $rest$ is empty. In this case, we need to add an extra node with the prefix key that will be a child to the last node of the path.

5. Key and $rest$ are non-empty strings. In this case, the node x_n is split into three ones, with prefixes $common$, $rest$ and key respectively, the last one is marked as $final$.

The tricky part is related to splitting of a block on steps 4 and 5. If the vertex which is being split is a leaf node of a twig that has a descendant twig we create a new vertex with a prefix key in a new twig. We need to find a child twig which lies in a block which has more free space left. In theory this is a very expensive operation because it affects all the blocks that contain the child twigs. The number of such blocks is not greater than the number of children twigs. But in practice such an approach is unacceptable since in addition to the compactness of storage the number of blocks being accessed should also be minimized. There is a compromise that allows to meet both requirements in practice: we limit the number of blocks accessed by some constant D (in our implementation $D = 2$). If none of the D blocks have enough space for the new vertex to be added, the block splitting procedure starts and after that the insert procedure should be executed again. By mean of this approach we will access not more than $D+2$ blocks (excluding the possible block splitting).

3.3.3 Block splitting

One of the distinguishing features of BST compared to B-trees is that the BST tree is not balanced. As we have proven earlier1 T is uniquely determined by the set K of stored strings so we can not decrease the height of the tree. Nevertheless there are studies that show that unbalanced tree affect the performance very little ([12]). The height of BST is the height of its twigs which must be read to find given prefix in the worst case.

The procedure of new space allocation performs the splitting of the block and is called only in case the block doesn't have enough space to insert a new vertex. We use two different strategies for block splitting.

The first strategy makes the blocks grow in "width" and doesn't affect the height of the tree so it is more preferable. The algorithm handles the situation when one block contains several twigs. In this case we divide twigs contained in block into two sets P_1 and P_2 such that $|\sum_{w \in P_1} w(B) - \sum_{w \in P_2} w(B)|$ is minimal over all possible partitions P_1 and P_2. Thus these sets must separate twigs contained in the block to about half of their total size. One of these sets remains in its original block, and a second set is moved to the new block. This algorithm affects exactly three blocks — one block that contains the parent node (this is a significant fact, which will play a role in calculating the total number of units affected by the separation.) and two blocks containing the twigs (old and new).

The second strategy is used in case the block where the vertex is added to has only one twig and there is still not enough space. In this case we are looking for the first node x starting from the root of the twig which has $N > 1$ links. This node has N child nodes that will be the roots of N new twigs. Next, we move all nodes from the root to x to the a parent twig contained in another block[4]. Next, we split the remaining N resulting twigs using the first split strategy. If the initial twig is the root of the tree, then the set of nodes is not transferred to the block that contains an parent vertex node (since it does not exist) but into the new block. This is the only case when the tree grows in height. This algorithm accesses exactly three blocks; one of these blocks contains the ancestor twig and two blocks contain the newly created twigs.

So the only case left to handle is the one when the block contains only one twig, and the first algorithm is applicable and at the same time in the block containing the branch of an ancestor there is not enough space to move the set of nodes of the twigs contained in the block that we are trying to separate.

While looking for a key during the insertion process, we build a path to the vertex which is going to be split during insertion. We collect some auxiliary data about blocks on our way: how many twigs do these blocks contain, how much space is occupied by the root nodes, how much free space do these blocks have, etc. It is very im-

[4]Strictly speaking there is no guarantee that the block that contains a parent twig has enough room to accept this set of nodes.

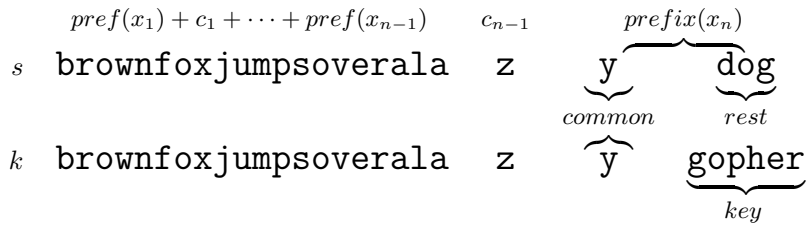

Figure 3: Example for insert

portant to note that we do not evaluate this data every time. It is stored in a block header and is updated only when the block is being changed.

With all this information collected we build the block splitting sequence starting from the block we need to insert new vertex to. The number of accessed blocks to $2h + 1$ in worst case where h is the height of the tree.

3.3.4 Deletion

Following the example of most papers on B-trees, we introduce an operation of *lazy removal*[10]. This approach is widely applied to the database systems because the procedure for the deletion of the element from the B-tree can be even trickier than the insert procedure. Lazy removal allows to significantly accelerate updates on the database.

In the developed structure delete is simply removing the flag $final$ from the vertex, that corresponds to the string we are looking for. Thus after the removal of the vertex we have the redundant vertex and consequently a non-minimal tree. This approach requires a procedure to minimize the tree (or a separate twig). We remove all redundant vertices, which in turn are of two types:

1. Vertices that are not marked as $final$ and have no outgoing edges. These nodes can be simply removed.

2. Vertices that are not marked as $final$ and have exactly one outgoing edge. In this case it is enough to merge the nodes with the concatenation of $prefix + edgecharacter + prefix$.

After this procedure is applied there are no redundant nodes remaining. It may also happen that some twigs have no nodes or only a single external-node remains. Such twigs must be removed and the child twigs are propagaded. In some cases this may require splitting of the block.

Thus minimizing the procedure for one twig may access 4 blocks in the worst case: 3 blocks by separation procedure of the original block and 1 block containing the descendant twig may be accessed.

3.4 Tests and results

Tests were performed within Sedna DBMS. All the tests were made on loaded DBLP Computer Science Bibliography[11] using B+ tree structure and BST both implemented in Sedna for handling indexes. Requests for the construction of the index are as follows:

For B+ tree:

```
> create index "urlbtree[1]"
  on doc("dblp.xml")/dblp/*
  by ./url as xs:string
  using "btree"&
```

For BST:

```
> create index "urlbstrie[1]"
  on doc("dblp.xml")/dblp/*
  by ./url as xs:string
  using "bstrie"&
```

In our tests we measure space occupied by the same indexes for both structures, the time needed for updates and search time. Indexes were created on publications by their identifiers, URI and EE field. The total number of indexed publications is about 3 million. However we show here only the space occupied by index, because the search time is almost the same for both types of trees.

	Distinct values	Key volume Kb	BST Kb	B-tree Kb
key	2 989 811	59 337	98 752	106 432
url	1 915 933	72 828	81 984	103 296
ee	1 667 275	74 975	53 440	101 248

Table 1: Comparison between BST and B+ tree

As one can see, BST can archive even significant data compression, while times of search and update are almost the same for both structures.

3.5 Conclusion

We have introduced a new trie structure that effectively solves the dictionary problem for strings of unlimited length and operates with large blocks of data.

Structure is implemented and is ready to use within Sedna XML DBMS[15]. The code is open source under Apache license and is available on the project site.

Tests show that the structure is well suited for use with indexes on fields that contain URI or strings which may contain the same prefixes. The BST structure also shows good results in the general case.

References

[1] Nikolas Askitis and Justin Zobel. B-tries for disk-based string management. *VLDB J.*, 18(1):157–179, 2009.

[2] Ricardo A. Baeza-Yates. An adaptive overflow technique for b-trees. In François Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *EDBT*, volume 416 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 1990.

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

[4] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[5] Walter A. Burkhard. Hashing and trie algorithms for partial match retrieval. *ACM Trans. Database Syst.*, 1(2):175–187, 1976.

[6] Eugene Inseok Chong, Jagannathan Srinivasan, Souripriya Das, Chuck Freiwald, Aravind Yalamanchi, Mahesh Jagannath, Anh-Tuan Tran, Ramkumar Krishnan, and Richard Jiang. A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary b^+-trees. *SIGMOD Record*, 32(2):78–88, 2003.

[7] Paolo Ferragina and Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[8] Andrey Fomichev, Maxim Grinev, and Sergei D. Kuznetsov. Sedna: A native xml dbms. In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2006.

[9] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[10] Theodore Johnson and Dennis Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *J. Comput. Syst. Sci.*, 47(1):45–76, 1993.

[11] Michael Ley. Die trierer informatik-bibliographie dblp. In *GI Jahrestagung*, pages 257–266, 1997.

[12] Joong Chae Na and Kunsoo Park. Simple implementation of string b-trees. In Alberto Apostolico and Massimo Melucci, editors, *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 214–215. Springer, 2004.

[13] Ratko Orlandic and Hosam M. Mahmoud. Storage overhead of o-trees, b-trees and prefix b-trees: A comparative analysis. *Int. J. Found. Comput. Sci.*, 7(3):209–226, 1996.

[14] Neal Sample, Brian F. Cooper, Michael J. Franklin, Gísli R. Hjaltason, Moshe Shadmon, and Levy Cohe. Managing complex and varied data with the indexfabric(tm). In *ICDE*, pages 492–493. IEEE Computer Society, 2002.

[15] Sedna XML DBMS source code downloads, 2012. http://sedna.org/download.html.

[16] B. Srinivasan. An adaptive overflow technique to defer splitting in b-trees. *Comput. J.*, 34(5):397–405, 1991.

[17] Wojciech Szpankowski. Patricia tries again revisited. *J. ACM*, 37(4):691–711, 1990.

[18] Ilya Taranov, Ivan Shcheklein, Alexander Kalinin, Leonid Novak, Sergei Kuznetsov, Roman Pastukhov, Alexander Boldakov, Denis Turdakov, Konstantin Antipin, Andrey Fomichev, Peter Pleshachkov, Pavel Velikhov, Nikolai Zavaritski, Maxim Grinev, Maria P. Grineva, and Dmitry Lizorkin. Sedna: native xml database management system (internals overview). In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD Conference*, pages 1037–1046. ACM, 2010.

[19] Nikolaus Walczuch and Herbert Hoeger. Using individual prefixes in b^+-trees. *Journal of Systems and Software*, 47(1):45–51, 1999.