Efficient Subset and Superset Queries

Iztok SAVNIK
Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Glagoljaška 8, 5000 Koper, Slovenia

Abstract. The paper presents index structure for storing and querying sets called $SetTrie$. Besides the operations $insert$ and $search$ defined for ordinary tries, we introduce the operations for retrieving subsets and supersets of a given set from $SetTrie$ tree. The performance of operations is analysed empirically in a series of experiments. The analysis shows that sets can be accessed in $\mathcal{O}(c * |set|)$ time where $|set|$ represents the size of parameter set. The constant c is up to 5 for subset case and approximately 150 in average case for the superset case.

Keywords. Containment queries, indexes, access methods, databases

Introduction

Set containment queries are common in applications based on object-oriented or object-relational database systems. Relational tables or objects from collections can have *set-valued attributes* i.e. the attributes that range over sets. Set containment queries can express either selection or join operation based on set containment condition [5,10,3].

In this paper we propose an index structure $SetTrie$ that implements efficiently the basic two types of set containment queries: subset and superset queries. We give the presentation of the proposed data structure, the operations defined on $SetTrie$ and thorough empirical analysis.

Let us first give a description of subset and superset operations in more detail. Let U be a set of ordered symbols. The subsets of U are denoted as *words*. Given a set of words S and a subset of U named X, we are interested in the following queries.

1. Is X a subset of any element from S?
2. Is X a superset of any element from S?
3. Enumerate all Y in S such that X is a subset of Y.
4. Enumerate all Y in S such that X is a superset of Y.

$SetTrie$ is a tree data structure similar to $trie$ [6]. The possibility to extend the performance of usual $trie$ from membership operation to subset and superset operations comes from the fact that we are storing *sets* and not the *sequences* of symbols as for ordinary tries. In the case of sets the ordering of symbols in a set is not important as it is in the case of text. As it will be presented in the paper the ordering of set elements can be exploited for the efficient implementation of containment operations.

We analyse subset and superset operations in two types of experiments. Firstly, we examine the execution of the operations on real-world data where sets represents words from the English dictionary. Secondly, we have tested the operations on artificially gen-

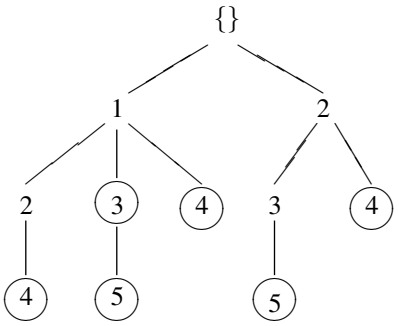

Figure 1. Example of $SetTrie$

erated data. In these experiments we tried to see how three main parameters: the size of words, the size of $SetTrie$ tree and the size of test-set, affect the behavior of the operations.

The paper is organized as follows. The following section presents the data structure $SetTrie$ together with the operations for searching the subsets and supersets in a tree. The Section 2 describes the empirical study of $SetTrie$. We present a series of experiments that measure the behavior of operations and the size of data structure. The related work is presented in Section 3. We give the presentation of existent work on set-valued attributes and containment queries as well as related work on trie and Patricia tree data structures. Finally, the overview and conclusions are given in Section 4.

1. Data Structure $SetTrie$

$SetTrie$ is a tree composed of nodes labeled with indices from 1 to N where N is the size of the alphabet. The root node is labeled with $\{\}$ and its children can be the nodes labeled from 1 to N. A root node alone represents an empty set. A node labeled i can have children labeled with numbers greater than i. Each node can have a flag denoting the last element in the set. Therefore, a set is represented by a path from the root node to a node with flag set to true.

Let us give an example of $SetTrie$. Figure 1 presents a $SetTrie$ containing the sets $\{1,3\}, \{1,3,5\}, \{1,4\}, \{1,2,4\}, \{2,4\}, \{2,3,5\}$. Note that flaged nodes are represented with circles.

Since we are dealing with sets for which the ordering of the elements is not important, we can define a syntactical order of symbols by assigning each symbol a unique index. Words are ordered by sequences of indices. The ordering of words is exploited for the *representation* of sets of words as well as in the *implementation* of the above stated operations.

$SetTrie$ is a tree storing a set of words which are represented by a path from the root of $SetTrie$ to a node corresponding to the indices of elements from words. As with tries, prefixes that overlap are represented by a common path from the root to an internal vertex of $SetTrie$ tree.

The operations for searching subsets and supersets of a set X in S use the ordering of U. The algorithms do not need to consider the tree branches for which we know they

do not lead to results on the basis of the ordering of word symbols. The search space for a given X and tree representing S can be seen as a subtree determined primarily by the search word X but also with the search tree corresponding to S.

1.1. Operations

Let us first present a data structure for storing *words*, that is, the sets of symbols. Words are stored in a data structure $Word$ representing ordered sets of integer numbers.

The users of $Word$ can scan sets using the following mechanism. The operation $word.gotoFirstElement$ sets the current element of word to the first element of ordered set. Then, the operation $word.existsCurrentElement$ checks if word has the current element set. The operation $word.currentElement$ returns the current element, and the operation $word.gotoNextElement$ goes to the next element in the set.

Let us now describe the operations of the data structure $SetTrie$. The first operation is insertion. The operation $insert$(root,word) enters a new $word$ into the $SetTrie$ referenced by the root $node$. The operation is presented by Algorithm 1.

Algorithm 1 insert($node$, $word$)

```
 1: if (word.existsCurrentElement) then
 2:    if (exists child of node labeled word.currentElement) then
 3:       nextNode = child of node labeled word.currentElement;
 4:    else
 5:       nextNode = create child of node labeled word.currentElement;
 6:    end if
 7:    insert(nextNode, word.gotoNextElement)
 8: else
 9:    node's flag_last = true;
10: end if
```

Each invocation of operation $insert$ either traverses through the existing tree nodes or creates new nodes to construct a path from the root to the flagged node corresponding to the last element of the ordered set.

The following operation $search$(node,word) searches for a given $word$ in the tree $node$. It returns true when it finds all symbols from the word, and false as soon one symbol is not found. The algorithm is shown in Algorithm 2. It traverses the tree $node$ by using the elements of ordered set $word$ to select the children.

Let us give a few comments to present the algorithm in more detail. The operation have to be invoked with the call $search$(root,set.gotoFirstElement) so that $root$ is the root of the $SetTrie$ tree and the current element of the $word$ is the first element of $word$. Each activation of $search$ tries to match the current element of $word$ with the child of $node$. If the match is not successful it returns $false$ otherwise it proceeds with the following element of $word$.

The operation $existsSubset$(node,word) checks if there exists a subset of $word$ in the given tree referenced by $node$. The subset that we search in the tree has fewer elements than $word$. Therefore, besides that we search for the exact match we can also skip one or more elements in $word$ and find a subset that matches the rest of the elements of $word$. The operation is presented in Algorithm 3.

Algorithm 2 search($node$, $word$)

```
1:  if ($word$.existsCurrentElement) then
2:     if (there exists child of $node$ labeled $word$.currentElement) then
3:        matchNode = child vertex of $node$ labeled $word$.currentElement;
4:        search($matchNode$, $word$.gotoNextElement);
5:     else
6:        return false;
7:     end if
8:  else
9:     return ($node$'s last_flag == true) ;
10: end if
```

Algorithm 3 existsSubset(node,set)

```
1:  if ($node$.last_flag == true) then
2:     return true;
3:  end if
4:  if (not $word$.existsCurrentElement) then
5:     return false;
6:  end if
7:  found = false;
8:  if ($node$ has child labeled $word$.currentElement) then
9:     $nextNode$ = child of $node$ labeled $word$.currentElement;
10:    $found$ = existsSubset($nextNode$, $word$.gotoNextElement);
11: end if
12: if (!found) then
13:    return existsSubset($node$,$word$.gotoNextElement);
14: else
15:    return true;
16: end if
```

Algorithm 3 tries to match elements of $word$ by descending simultaneously in tree and in $word$. The first IF statement (line 1) checks if a subset of $word$ is found in the tree i.e. the current node of a tree is the last element of subset. The second IF statement (line 4) checks if $word$ has run of the elements. The third IF statement (line 8) verifies if the parallel descend in $word$ and tree is possible. In the positive case, the algorithm calls $existsSubset$ with the next element of $word$ and a child of $node$ corresponding to matched symbol. Finally, if match did not succeed, current element of $word$ is skipped and $existsSubset$ is activated again in line 13.

The operation $existsSubset$ can be easily extended to find all subsets of a given $word$ in a tree $node$. After finding the subset in line 15 the subset is stored and the search continues in the same manner as before. The experimental results with the operation $getAllSubsets$(nod,word) are presented in the following section.

The operation $existsSuperset$(node,word) checks if there exists a superset of $word$ in the tree referenced by $node$. While in operation $existsSubset$ we could skip some elements from $word$, here we can do the opposite: the algorithm can skip some elements

Algorithm 4 existsSuperset($node$, $word$)

```
1:  if (not word.existsCurrentElement) then
2:     return true;
3:  end if
4:  found = false;
5:  from = word.currentElement;
6:  upto = word.nextElement if it exists and N otherwise;
7:  for (each child of node labeled l: from < l ≤ upto) while !found do
8:     if (child is labeled upto) then
9:        found = existsSuperset(child,word.gotoNextElement);
10:    else
11:       found = existsSuperset(child,word);
12:    end if
13: end for
```

in supersets represented by $node$. Therefore, $word$ can be matched with the subset of superset from a $tree$. The operation is presented in Algorithm 4

Let us present Algorithm 4 in more detail. The first IF statement checks if we are already at the end of $word$. If so, then the parameter $word$ is covered completely with a superset from $tree$. Lines 5-6 set the lower and upper bounds of iteration. In each pass we either skip current $child$ and call $existsSuperset$ on unchanged $word$ (line 11), or, descend in parallel on both $word$ and tree in the case that we reach the upper bound ie. the next element in $word$ (line 9).

Again, the operation $existsSuperset$ can be quite easily extended to retrieve all supersets of a given $word$ in a tree $node$. However, after $word$ (parameter) is matched completely (line 2 in Algorithm 4), there remains a subtree of trailers corresponding to a set of supersets that subsume $word$. This subtree is rooted in a tree node, let say $node_k$, that corresponds to the last element of $word$. Therefore, after the $node_k$ is matched against the last element of the set in line 2, the complete subtree has to be traversed to find all supersets that go through $node$.

2. Experiments

The performance of the presented operations is analysed in four experiments. The main parameters of experiments are: the number of words in the tree, the size of the alphabet, and the maximum length of words. The parameters are named: $numTreeWord$, $alphabetSize$, and $maxSizeWord$, respectively. In every experiment we measure the *number of visited nodes necessary for an operation to terminate*.

In the first experiment, $SetTrie$ is used to store real-world data – it stores the words from English Dictionary. In the following three experiments we use artificial data – datasets and test data are randomly generated. In these experiments we analyse in detail the interrelations between one of the stated tree parameters on the number of visited nodes.

In all experiments we observe four operations presented in the previous section: $existsSubset$ (abbr. esb) and its extension $getAllSubsets$ (abbr. gsb), and $existsSuperset$ (abbr. esr) and its extension $getAllSupersets$ (abbr. gsr).

2.1. Experiment with Real-World Data

Table 1. Visited nodes for dictionary words

word length	esr	gsr	esb	gsb
2	523	169694	1	1
3	3355	103844	3	3
4	12444	64802	6	6
5	9390	34595	11	12
6	11500	22322	14	19
7	12148	17003	18	32
8	8791	10405	19	46
9	6985	7559	19	78
10	3817	3938	21	102
11	3179	3201	20	159
12	2808	2820	20	221
13	2246	2246	22	290
14	1651	1654	19	403
15	1488	1488	18	575
16	895	895	19	778
17	908	908	20	925
18	785	785	18	1137
19	489	489	22	1519
20	522	522	19	1758
21	474	474	19	2393
22	399	399	17	3044
23	362	362	17	3592
24	327	327	19	4167

Let us now present the first experiment in more detail. The number of words in test set is 224,712 which results in a tree with 570,462 nodes. The length of words are between 5 and 24 and the size of the alphabet ($alphabetSize$) is 25. The test set contains 10,000 words.

Results are presented in Table 1 and Figure 2. Since there are 10,000 words and 23 different word lengths in the test set, approximately 435 input words are of the same length. Table 1 and Figure 2 present the average number of visited nodes for each input word length (except for gsr where values below word length 6 are intentionally cut off).

Let us give some comments on the results presented in Table 1. First of all, we can see that the superset operations (esr and gsr) visit more nodes than subset operations (esb and gsb).

The number of nodes visited by esr and gsr decreases as the length of words increases. This can be explained by more constrained search in the case of longer words, while it is very easy to find supersets of shorter words and, furthermore, there are a lot of supersets of shorter words in the tree.

Since operation gsr returns all supersets (of a given set), it always visits more nodes than the operation esr. However, searching for the supersets of longer words almost always results in failure and for this reason the number of visited nodes is the same for both operations.

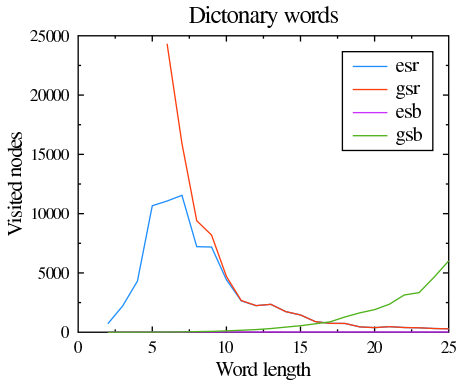

Figure 2. Number of visited nodes

The number of visited nodes for esb in the case that words have more than 5 symbols is very similar to the length of words. Below this length of words both esb and gsb visit the same number of nodes, because there were no subset words of this length in the tree and both operations visit the same nodes.

The number of visited nodes for gsb linearly increases as the word length increases. We have to visit all the nodes that are actually used for the representation of all subsets of a given parameter set.

2.2. *Experiments with Artificial Data*

In $experiment1$ we observe the influence of changing the maximal length of word to the performance of all four operations. We created four trees with $alphabetSize$ 30 and $numTreeWord$ 50,000. $maxSizeWord$ is different in each tree: 20, 40, 60 and 80, for tree1, tree2, tree3 and tree4, respectively. The length of word in each tree is evenly distributed between the minimal and maximal word size. The number of nodes in the trees are: 332,182, 753,074, 1,180,922 and 1,604,698. The test set contains 10,000 words.

Figure 3 shows the performance of all four operations on all four trees. The performance of superset operations is affected more by the change of the word length than the subset operations.

With an even distribution of data in all four trees, esr visits most nodes for input word lengths that are about half of the size of $maxSizeWord$ (as opposed to dictionary data where it visits most nodes for word lengths approximately one fifth of $maxSizeWord$). For word lengths equal to $maxSizeWord$ the number of visited nodes is roughly the same for all trees, but that number increases slightly as the word length increases.

esb operation visits fewer than 10 nodes most of the time, but for $tree3$ it goes up to 44 which is still a very low number. The experiment was repeated multiple (about 10) times, and in every run the operation "jumped up" in a different tree. As seen later in $experiment2$, it seems that $numTreeWord$ 50 is just on the edge of the value where esb stays constantly below 10 visited nodes. It is safe to say that the change in $maxSizeWord$ has no major effect on $existsSubSet$ operation.

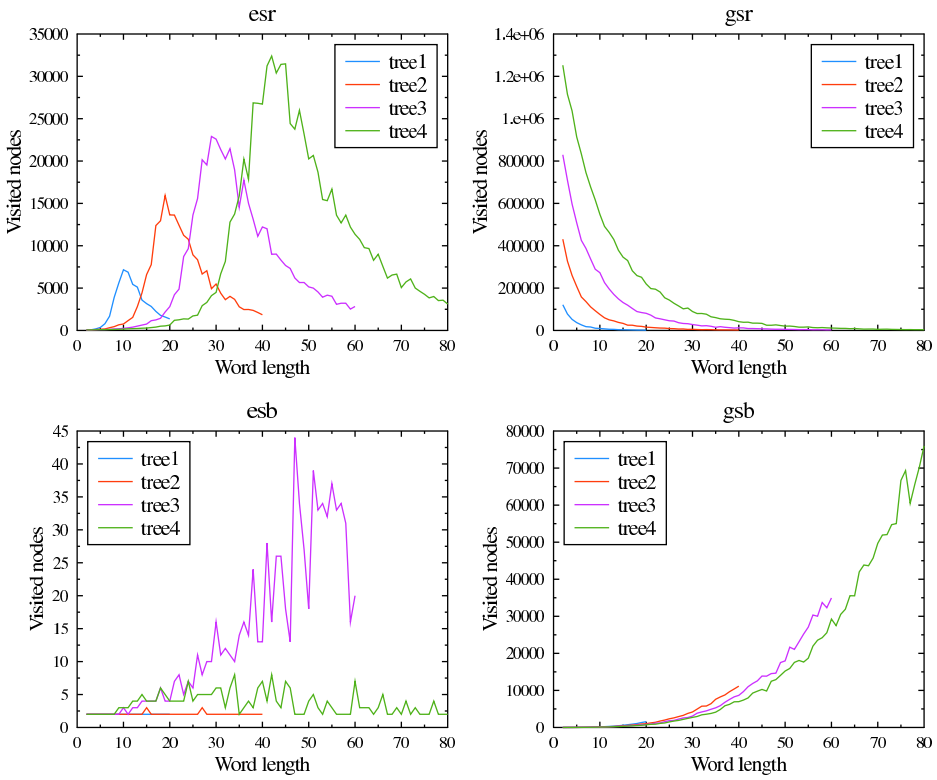

Figure 3. Experiment 1 – increasing $maxSizeWord$

In contrast to gsr, gsb visits less nodes for the same input word length in trees with greater $maxSizeWord$, but the change is minimal. For example for word length 35 in $tree2$ ($maxSizeWord$ 40) gsb visits 7,606 nodes, in $tree3$ ($maxSizeWord$ 60) it visits 5,300 nodes and in $tree4$ ($maxSizeWord$ 80) it visits 4,126 nodes.

In $experiment2$ we are interested about how a change in the number of words in the tree affects the operations. Ten trees are created all with $alphabetSize$ 30 and $maxSizeWord$ 30. $numTreeWord$ is increased in each tree by 10,000 words: $tree1$ has 10,000 words, and $tree10$ has 100,000 words. The number of nodes in the trees (from $tree1$ to $tree10$) are: 115,780, 225,820, 331,626, 437,966, 541,601, 644,585, 746,801, 846,388, 946,493 and 1,047,192. The test set contains 5,000 words.

Figure 4 shows the number of visited nodes for each operation on four trees: $tree1$, $tree4$, $tree7$ and $tree10$ (only every third tree is shown to reduce clutter). When increasing $numTreeWord$ the number of visited nodes increases for esr, gsr and gsb operations. esb is least affected by the increased number of words in the tree. In contrast to the other three operations, the number of visited nodes decreases when $numTreeWord$ increases.

For input word lengths around half the value of $maxSizeWord$ (between 13 and 17) the number of visited nodes for esr increases with the increase of the number of words in the tree. For input word lengths up to 10, the difference between trees is minimal.

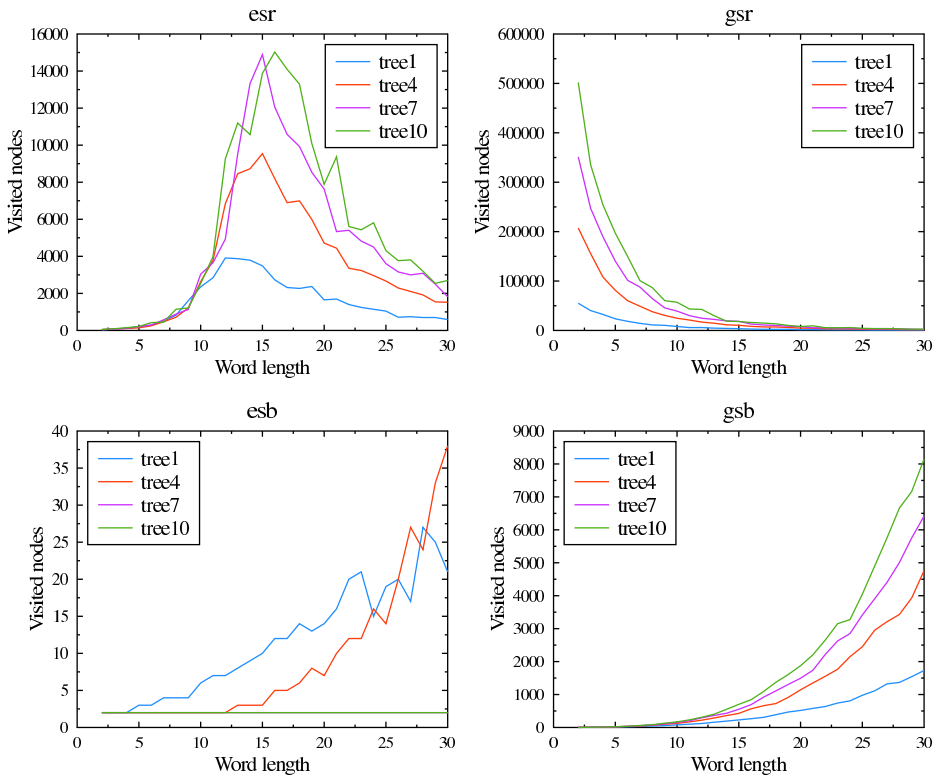

Figure 4. Experiment 2 – increasing $numTreeWord$

After word lengths about 20 the difference in the number of visited nodes between trees starts to decline. Also, trees 7 to 10 have very similar results. It seems that after a certain number of words in the tree the operation "calms down".

The increased number of words in the tree affects the gsr operation mostly in the first quarter of $maxSizeWord$. The longer the input word, the lesser the difference between trees. Still, this operation is the most affected by the change of $numTreeWord$. The average number of visited nodes for all input word lengths in tree1 is 8,907 and in tree10 it is 68,661. Due to the nature of the operation, this behavior is expected. The more words there are in the tree, the more supersets can be found for an input word.

As already noted above, when the number of words in the tree increases the number of visited nodes for esb decreases. After a certain number of words, in our case this was around 50,000, the operation terminates at a minimum possible visits of nodes for any word length. The increase of $numTreeWord$ seems to "push down" the operation from left to right. This can be seen in figure 4 by comparing $tree1$ and $tree4$. In $tree1$ the operation visits more then 10 after word length 15, and in $tree4$ it visits more than 10 nodes after word length 23. Overall the number of visited nodes is always very low.

The chart of gsb operation looks like a mirrored chart of gsr. The increased number of words in the tree has more effect on input word lengths where the operation visits more nodes (longer words). Below word length 15 the difference between trees is in the

range of 100 visited nodes. At word length 30 gsb visits 1,729 nodes in $tree1$ and 8,150 nodes in $tree10$. The explanation in for the increased number of visited nodes is similar as for gsr operation: the longer the word, the more subsets it can have, the more words in the tree, the more words with possible subsets there are.

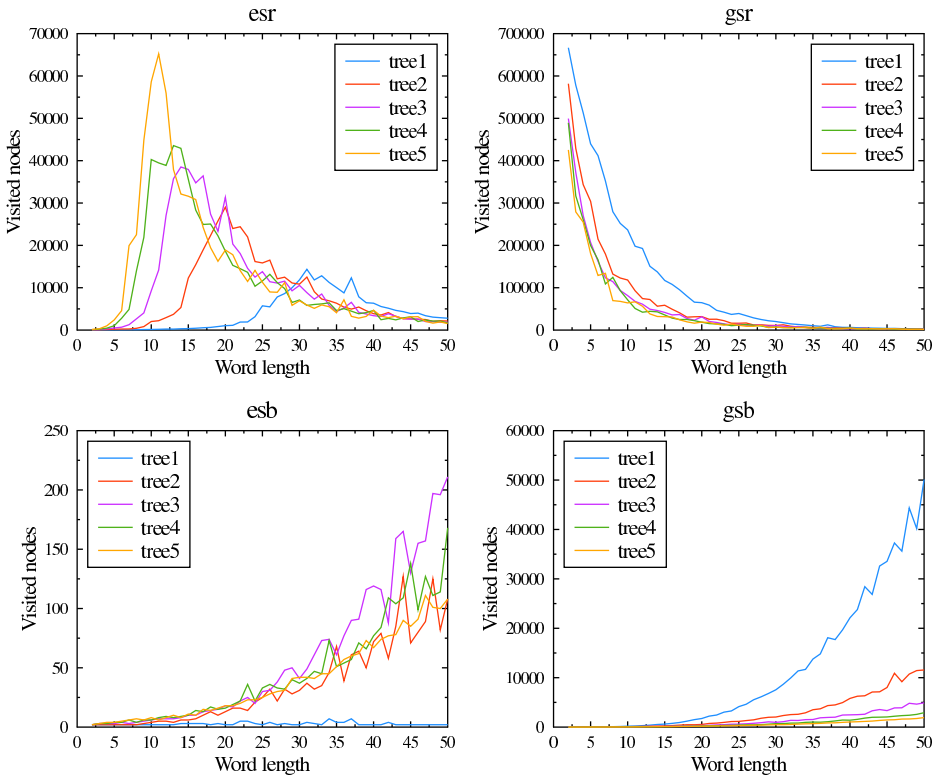

Figure 5. Experiment 3 – increasing $alphabetSize$

In $experiment3$ we are interested about how a change in the alphabet size affects the operations. Five trees are created with $maxSizeWord$ 50 and $numTreeWord$ 50,000. $alphabetSize$ is 20, 40, 60, 80 and 100, for $tree1$, $tree2$, $tree3$, $tree4$ and $tree5$, respectively. The number of nodes in the trees are: 869,373, 1,011,369, 1,069,615, 1,102,827 and 1,118,492. The test set contains 5,000 words.

When increasing $alphabetSize$ the tree becomes sparser–the number of child nodes of a node is larger, but the number of nodes in all five trees is roughly the same. For gsr and more notably gsb operation, visit less nodes for the same input word length: the average number of visited nodes decreased when $alphabetSize$ increases. The esr operation on the other hand visits more nodes in trees with larger alphabetSize.

The number of visited nodes of esr increases with the increase of $alphabetSize$. This is because it is harder to find supersets of given words, when the number of symbols that make up words is larger. The effect is greater on word lengths below half $maxSizeWord$. The number of visited nodes starts decreasing rapidly after a certain word length. At this point the operation does not find any supersets and it returns false.

gsr is not affected much by the change of $alphabetSize$. The greatest change happens when increasing $alphabetSize$ over 20 ($tree1$). The number of visited nodes in trees 2 to 5 is almost the same, but it does decrease with every increase of $alphabetSize$.

In $tree1$ *esb* visits on average 3 nodes. When we increase $alphabetSize$ the number of visited nodes also increases, but as in gsr the difference between trees 2 to 5 is small.

The change of $alphabetSize$ has a greater effect on longer input words for the gsr operation. The number of visited nodes decreased when $alphabetSize$ increased. Here again the biggest change is when going over $alphabetSize$ 20. With every next increase, the difference in the number of visited nodes is smaller.

3. Related work

The initial implementation of $SetTrie$ was in the context of a datamining tool $fdep$ which is used for the induction of functional dependencies from relations [8,9]. $SetTrie$ serves there for storing and retrieving hypotheses that basically correspond to *sets*.

The data structure we propose is similar to trie [6,7]. Since we are not storing sequences but *sets* we can exploit the fact that the order in sets is not important. Therefore, we can take advantage of this to use syntactical order of elements of sets and obtain additional functionality of tries.

Sets are among important data modeling constructs in object-relational and object-oriented database systems. *Set-valued attributes* are used for the representation of properties that range over sets of atomic values or objects. Database community has shown significant interest in indexing structures that can be used as access paths for querying set-valued attributes [10,5,3,11,12].

Set containment queries were studied in the frame of different index structures. Helmer and Moercotte investigated four index structures for querying set-valued attributes of low cardinality [3]. All four index structures are based on conventional techniques: signatures and inverted files. Index structures compared are: sequential signature files, signature trees, extendable signature hashing, and B-tree based implementation of inverted lists. Inverted file index showed best performance over other data structures in most operations.

Zhang et al. [12] investigated two alternatives for the implementation of containment queries: a) separate IR engine based on inverted lists and b) native tables of RDBMS. They have shown that while RDBMS are poorly suited for containment queries they can outperform inverted list engine in some conditions. Furthermore, they have shown that with some modifications RDBMS can support containment queries much more efficiently.

Another approach to the efficient implementation of set containment queries is the use of signature-based structures. Tousidou et al. [11] combine the advantages of two access paths: linear hashing and tree-structured methods. They show through the empirical analysis that S-tree with linear hash partitioning is efficient data structure for subset and superset queries.

From the other perspective, our problem is similar to searching substrings in strings for which $tries$ and *Suffix trees* can be used. Firstly, Rivest examines [6] the problem of partial matching with the use of hash functions and $trie$ trees. He presents an algorithm for partial match queries using $tries$. However, he does not exploit the ordering of indices that can only be done in the case that *sets* are stored in tries.

Baeza-Yates and Gonnet present an algorithm [1] for searching regular expressions using $Patricia$ trees as the logical model for the index. They simulate a finite automata over a binary Particia tree of words. The result of a regular expression query is a superset or subset of the search parameter.

Finally, Charikar et al. [2] present two algorithms to deal with a subset query problem. The purpose of their algorithms is similar to $existsSuperSet$ operation. They extend their results to a more general problem of orthogonal range searching, and other problems. They propose a solution for "containment query problem" which is similar to our 2. query problem introduced in Introduction.

4. Conclusions

The paper presents a data structure $SetTrie$ that can be used for efficient storage and retrieval of subsets or supersets of a given $word$. The performance of $SetTrie$ is shown to be efficient enough for manipulating sets of sets in practical applications.

Enumeration of subsets of a given universal set U is very common in *machine learning* [4] algorithms that search hypotheses space ordered in a lattice. Often we have to see if a given set, a subset or a superset has already been considered by the algorithm. Such problems include discovery of association rules, functional dependencies as well as some forms of propositional logic.

Finally, the initial experiments have been done to investigate if $SetTrie$ can be employed for searching substrings and superstrings in texts. Fur this purpose the data structure $SetTrie$ has to be augmented with the references to the position of words in text. While the data structure is relatively large "index tree", it may still be useful because of the efficient search.

References

[1] Baeza-Yates, R., Gonnet, G.: Fast text searching for regular expressions or automation searching on tries. *Journal of ACM* **43**(6) (1996), 915–936.

[2] Charikar, M., Indyk, P., Panigrahy, R.: New algorithms for subset query, partial match, orthogonal range searching and related problems. In: *Proc. 29th International Colloquium on Algorithms, Logic, and Programming*, LNCS **2380** (2002), 451–462.

[3] Helmer, S., Moerkotte, G.: A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal – The International Journal on Very Large Data Bases* **12**(3) (2003), 244–261.

[4] Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowl-edge discovery. *Data Mining and Knowledge Discovery Journal* **1**(3) (1997), 241–258.

[5] Melnik, S., Garcia-Molina, H.: Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems* **28**(2) (2003), 1–38.

[6] Rivest, R.: Partial-match retrieval algorithms. *SIAM Journal on Computing* **5**(1) (1976).

[7] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, Second Edition, MIT Press, 2001.

[8] Savnik, I., Flach, P.A.:, Bottom-up Induction of Functional Dependencies from Relations. In: *Proc. of KDD'93 Workshop: Knowledge Discovery from Databases*, AAAI Press, Washington (1993), 174–185.

[9] Flach, P.A., Savnik, I.: Database dependency discovery: a machine learning approach, *AI Communications* **12**(3) (1999), 139–160.

[10] Terrovitis, M., Passas, S., Vassiliadis, P., Sellis, T.: A Combination of trie-trees and inverted files for the indexing of set-valued attributes. In: *Proc. of ACM International Conference on Information and Knowledge Management* (2006).

[11] Tousidou, E., Bozanis, P., Manolopoulos, Y.: Signature-based structures for objects with set-valued attributes. *Information Systems* **27** (2002), 93–121.
[12] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: *ACM SIGMOD* (2001).