# Index Data Structure for Fast Subset and Superset Queries

Iztok Savnik



# Index data structure for fast subset and superset queries

Iztok Savnik

[1] Faculty of mathematics, natural sciences and information technologies,
University of Primorska, 6000 Koper, and
[2] Artificial Intelligence Laboratory, Jozef Stefan Institute, 1000 Ljubljana, Slovenia.
iztok.savnik@upr.si

**Abstract.** A new data structure *set-trie* for storing and retrieving sets is proposed. Efficient manipulation of sets is vital in a number of systems including datamining tools, object-relational database systems, and rule-based expert systems. Data structure *set-trie* provides efficient algorithms for set containment operations. It allows fast access to subsets and supersets of a given parameter set. The performance of operations is analyzed empirically in a series of experiments on real-world and artificial datasets. The analysis shows that sets can be accessed in $\mathcal{O}(c * |set|)$ time where $|set|$ represents the size of parameter set and $c$ is a constant.

**Keywords:** subset queries, set containment queries, partial matching, access methods, database index.

## 1 Introduction

*Set containment queries* are common in various systems including datamining tools, object-relational databases, rule-based expert systems, and AI planning systems. Enumeration of subsets of a given universal set $U$ is very common in *data mining* algorithms [10] where sets are used as basis for the representation of hypotheses and search space forms a lattice. Often we have to see if a given hypothesis has already been considered by the algorithm. This can be checked by searching the set of hypotheses (sets) that have already been processed. Furthermore, in some cases hypotheses can be easily overthrown if a superset hypothesis has already been shown not valid. Such problems include discovery of association rules, functional dependencies as well as some forms of propositional logic [10, 15, 5, 8].

In *object-relational database management systems* tables can have set-valued attributes i.e. attributes that range over sets. Set containment queries can express either selection or join operation based on set containment condition. Efficient access to relation records based on conditions that involve set operations are vital for fast implementation of such queries [12, 18, 7].

*Rule-based expert systems* use set containment queries to implement fast pattern-matching algorithms that determine which rules are fired in each cycle of expert system execution. Here sets form pre-conditions of rules composed of elementary conditions.

Given a set of valid conditions the set of fired rules includes those with pre-condition included in this set [6, 4].

Finally, in *AI planning systems* goal sets are used to store goals to be achieved from a given initial state. Planning modules use subset queries in procedure that examines if a given goal set is satisfiable. Part of the procedure represents querying goal sets that were previously shown to be unsatisfiable. Here also sets are used to form basic structure of hypothesis space [2].

In this paper we propose a novel index data structure *set-trie* that implements efficiently basic two types of set containment queries: *subset* and *superset queries*. *Set-trie* provides storage for sets as well as multisets. Preliminary version of this paper has been published in [16].

*Set-trie* is a tree data structure similar to $trie$ [13]. 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* (*multisets*) and not the *sequences* of symbols as for ordinary tries. In case of sets (multisets) 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 is used as the basis for the definition of efficient algorithms for set containment operations. Since the semantics of set containment operations is equivalent to the semantics of multiset containment operations we will in the following text sometimes refer to both, sets and multisets, as *sets*.

We analyze subset and superset operations in two types of experiments. Firstly, we examine the execution of the operations on real-world data where multisets represent words from the English dictionary. Secondly, we have tested the operations on artificially generated data. In these experiments we tried to see how three main parameters: the size of sets, the size of *set-trie* tree and the size of test-set, affect the behavior of the operations. Analysis shows that sets can be accessed in $\mathcal{O}(c * |set|)$ time where $|set|$ represents the size of parameter set and $c$ is a constant. The constant $c$ is up to $5$ for subset case and approximately $150$ in average case for the superset case.

The paper is organized as follows. The following section presents the data structure *set-trie* together with the operations for searching the subsets and supersets in a tree. The Section 3 describes the empirical study of *set-trie*. We present a series of experiments that measure the behavior of operations and the size of data structure. Related work is presented in Section 4. We give presentation of existent work from the fields of algorithms and data structures, AI systems where sets are used for querying hypotheses and states, and object-relational database systems where indexes are used to access set-valued attributes. Finally, the conclusions and the directions of our further work are given in Section 5.

## 2 Data structure *set-trie*

*Set-trie* 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 or equal $i$. 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 *set-trie*. Figure 2 presents a *set-trie* containing the sets $\{1,3\}, \{1,3,5\}, \{1,4\}, \{1,2,4\}, \{2,4\}, \{2,3,5\}$. Note that flagged nodes are represented with circles.

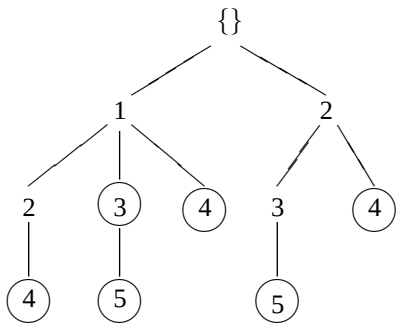


**Fig. 1.** Example of *set-trie*

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 sequences of symbols ordered by indices. The ordering of symbols is exploited for the *representation* of sets of words as well as in the *implementation* of the above stated operations.

*Set-trie* is a tree storing a set of words which are represented by a path from the root of *set-trie* 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 *set-trie* 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. 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 $S$.

### 2.1 Set containment operations

Let us first give formal definition of the basic subset and superset operations. 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) existsSubset$(S, X)$ returns $true$ if $\exists Y \in S : Y \subseteq X$ and $false$ otherwise.
2) existsSuperset$(S, X)$: returns $true$ if $\exists Y \in S : X \subseteq Y$ and $false$ otherwise.
3) getAllSubsets$(S, X)$: returns all sets $Y$ such that $Y \in S \wedge Y \subseteq X$.
4) getAllSupersets$(S, X)$: returns all sets $Y$ such that $Y \in S \wedge X \subseteq Y$.

Let us now present a data structure *Word* for storing sets of symbols. Symbols in words are represented as integer numbers. Elements of a set represented by $Word$ can be scanned 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.

Using data structure $Word$ we can now describe the operations of the data structure *set-trie*. The first operation is *insertion*. The operation $insert$(root,word) enters a new $word$ into the *set-trie* 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.

**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
```

Let us give a few comments to present the algorithm in more detail. The operation have to be invoked with the call $search$(root,word.gotoFirstElement) so that $root$ is the root of the *set-trie* 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 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 (parallel descend). Finally, if match did not succeed, current element of $word$ is skipped and $existsSubset$ is called with same $node$ and next element of $word$ 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 $getAllSubsets$(node,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 el-

**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 not 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
```

ements from $word$, here we can do the opposite: the algorithm can skip some elements 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 take 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 i.e. 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 $node_k$.

## 3 Experiments

The performance of the presented operations is analyzed in four experiments. The main parameters of experiments are: number of words in tree, size of the alphabet, and maximal 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, *set-trie* is used to store real-world data – it stores multisets obtained from words of English Dictionary. In the following three experiments we use artificial data – datasets include randomly generated sets of sets. In these experiments we analyze in detail the interrelations between one of the stated tree parameters and 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. $gsr$).

### 3.1 Experiment with real-world data

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).

| 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 |

**Fig. 2.** Visited nodes for dictionary words

Let us give some comments on the results presented in Table 2. 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.

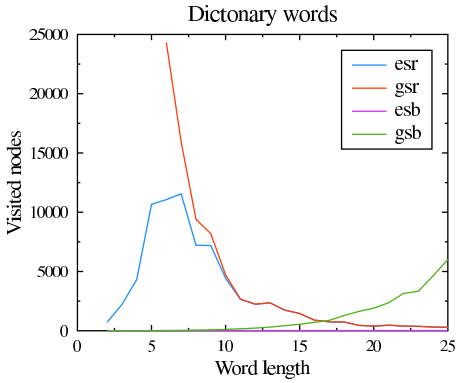


**Fig. 3.** Number of visited nodes

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.

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.

### 3.2 Experiments with artificial data

Three experiments were done by using artificially generated data. Experiments are named $experiment1$, $experiment2$ and $experiment3$.
1) In $experiment1$ we observe relation between maximal length of words to number of visited nodes of *set-trie* in all four operations.
2) $experiment2$ shows the relation between number of words stored in *set-trie* and number of visited nodes in *set-trie* in all four operations.
3) $experiment3$ investigates the relation between the size of alphabet and number of visited nodes of *set-trie* in all four operations.

Let us start with $experiment1$. Here we observe the influence of maximal length of words to the performance of all four operations. We created four trees with $alphabetSize$

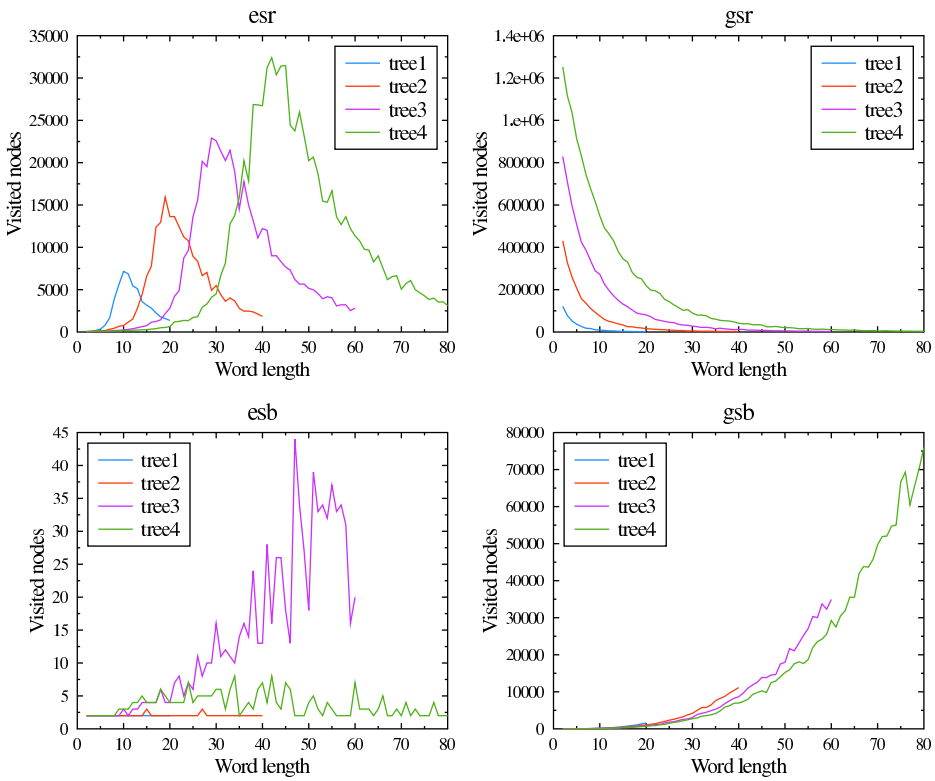


**Fig. 4.** Experiment 1 - increasing $maxSizeWord$

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 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. The experiment was repeated multiple (about 10) times, and in every run the operation jumped up in a different tree. As will be 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.

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.

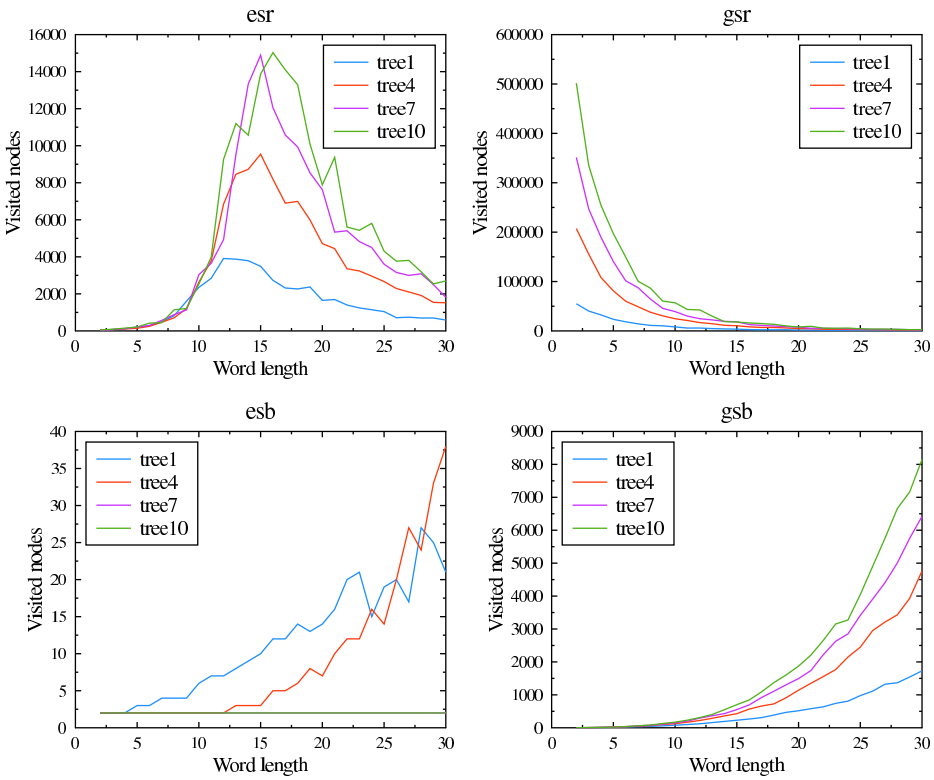


**Fig. 5.** Experiment 2 - increasing $numTreeWord$

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. $numTreeWord$ increases in each subsequent 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 in-

creasing $numTreeWord$ 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. 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 remains constant.

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 affected most 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 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, number of visited nodes for $esb$ decreases. After a certain number of words, in our case this was around 50,000, the operation terminates with minimal (possible) number of visited nodes for any word length. The increase of $numTreeWord$ pushes down the performance of 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 low.

The chart of $gsb$ operation looks like a mirrored chart of $gsr$. The increased number of words in 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 $tree10$. Explanation of 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.

In $experiment3$ we are interested about how a change of 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. Operation $gsr$ and more notably $gsb$ operation, visit less nodes for the same input word length: the average number of visited nodes decreased when $alphabetSize$ increases. Operation $esr$ on the other hand visits more nodes in trees with larger alphabetSize.

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. This is more evident for word lengths below half

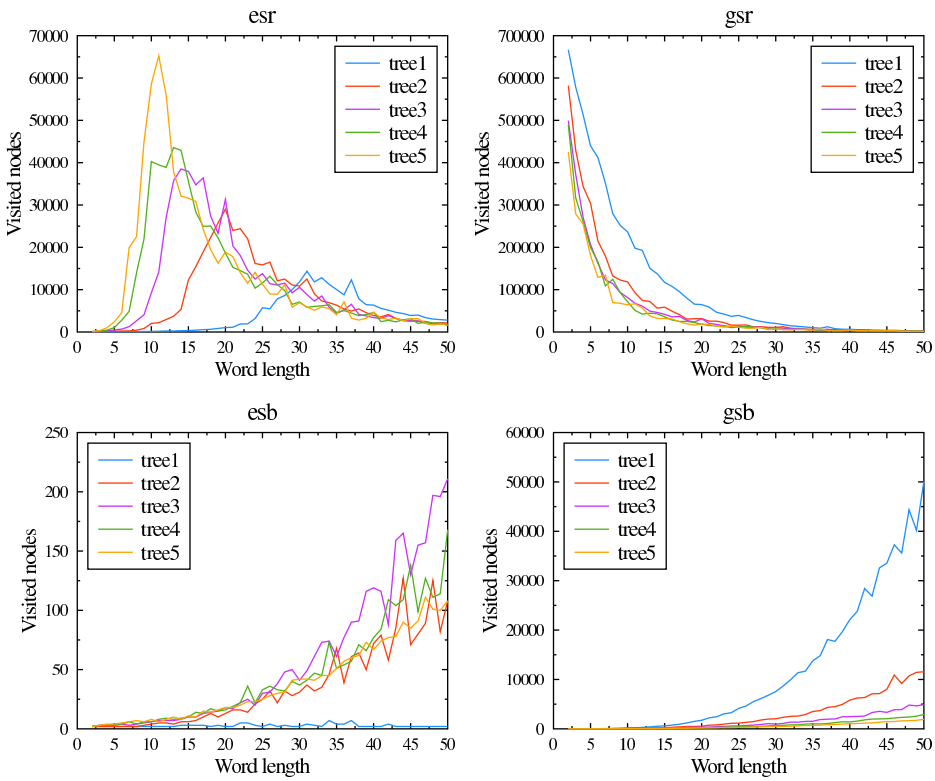


**Fig. 6.** Experiment 3 - increasing $alphabetSize$

$maxSizeWord$. 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.

Operation $gsr$ is not affected much by the change of $alphabetSize$. More evident change appears when $alphabetSize$ is increased over 20 ($tree1$). The number of visited nodes in trees 2 to 5 is almost the same, but it does decrease with the 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 more significant effect on longer input words for the $gsr$ operation. The number of visited nodes decreased when $alphabetSize$ increased. Here again the most evident change is when going over $alphabetSize$ 20. In each subsequent increase, the difference in the number of visited nodes is smaller.

## 4 Related work

The problem of querying sets of sets appears in various areas of Computer Science. Firstly, the problem has been studied in the form of *substring search* by Rivest [13],

Baeza-Yates [1] and Charikar [3]. Secondly, the subset queries are studied in various sub-areas of AI for storing and querying: pre-conditions of a large set of rules [6], states in planning for storing goal sets [8] and hypotheses in data mining algorithms [9]. Finally, querying sets is an important problem in object-relational databases management systems where attributes of relations can range over sets [18, 12, 7, 19, 20].

### 4.1 Partial-matching and containment query problem

The data structure we propose is similar to trie [13, 14]. 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.

Our problem is similar to searching substrings in strings for which *tries* and *Suffix trees* can be used. Firstly, Rivest examines [13] 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* or *multisets* 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. [3] 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 Section 1.

### 4.2 Querying hypotheses and states in AI systems

The initial implementation of *set-trie* was in the context of a datamining tool $fdep$ which is used for the induction of functional from relations [15, 5]. It has been further used in datamining tool $mdep$ for the induction of multivalued dependencies from relations [17]. In both cases sets are used as the basis for the representation of dependencies. Hypotheses (dependencies) are checked against the negative cover of *invalid dependencies* represented by means of *set-trie*. Furthermore, positive cover including valid dependencies is minimized by using *set-trie* as well.

Doorenbos in [4] proposes an index structure for querying pre-conditions of rules to be matched while selecting the next rule to activate in a rule-based system Rete [6]. Index structure stores conditions in separate nodes that are linked together to form pre-conditions of rules. Common conditions of rules are shared among the rules: lists of conditions with common prefix share all nodes that form prefix. Given a set of conditions that are fulfilled all rules that contain as pre-condition a subset of given set of conditions can be activated.

Similar index structure is proposed by Hoffman and Koehler by means of Unlimited Branching Tree (abbr. UBTree) to store set of sets. The main difference with the representation of rules in expert systems is that UBTree does not use variables. Children of

node are stored in a list attached to node. A set is in UBTree represented by a path from root to final node; path is labeled by elements of a set. The search procedures for subset and superset problems are similar to those we propose, however, the main difference in procedures is that we explicitly use ordering of sets for search while Hoffman and Koehler give a more general algorithm allowing other heuristic to be exploited. Our publication in 1993 [15] evidently presents the independence of work.

### 4.3 Indexing set-valued attributes of object-relational databases

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 [18, 12, 7, 19, 20].

*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 [7]. 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. [20] 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. [19] 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.

## 5 Conclusions

The paper presents a data structure *set-trie* that can be used for efficient storage and retrieval of subsets or supersets of a given $word$. The algorithms of set containment operations are analyzed empirically. It has been demonstrated that the algorithms are stable when used on real-world and artificially generated data. Empirical analysis was used to determine the behavior of each particular set containment operations. The performance of *set-trie* is shown to be efficient enough for storage and retrieval of sets in practical applications.

Initial experiments have been done to investigate if *set-trie* can be employed for searching substrings and superstrings in texts. For this purpose the data structure *set-trie* has to be augmented with the references to the position of words in text. As in the

case of indexes used in information retrieval [11] *set-trie* can be decomposed into *dictionary* and *postings*. Empirical analysis which would show memory consumption and efficiency of *set-trie* used for indexing huge quantities of texts remains to be completed.

## References


1. Baeza-Yates, R., Gonnet, G.: Fast text searching for regular expressions or automation searching on tries. Journal of ACM 1996, Vol.43, No.6, pp. 915-936.
2. Blurn, A., Furst, M., Fast planning through planning graph analysis, Artificial Intelligence, Vol.90, Issue 1-2, pp.279-298, 1997.
3. Charikar, M., Indyk, P., Panigrahy, R.: New Algorithms for Subset Query, Partial Match, Orthogonal Range Searching and Related Problems. LNCS 2002; Vol 2380, p. 451-462.
4. Doorenbos, R., Combining left and right unlinking for matching a large number of learned rules, AAAI-94, pp.451-458, 1994.
5. Flach, P.A., Savnik, I.: Database dependency discovery: a machine learning approach. AI Communications, Vol.12, No.3, IOS Press, 1999, pp.139-160.
6. Forgy, C., Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence, 19:17-37, 1982.
7. 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, Volume 12 Issue 3, 2003 pp. 244-261.
8. Hoffmann, J., Koehler, J., A New Method to Index and Query Sets, IJCAI, 1999.
9. Mamoulis, N., Cheung, D.W., Lian, W., Similarity Search in Sets and Categorical Data Using the Signature Tree, ICDE, 2003.
10. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery, Data Mining and Knowledge Discovery Journal, 1(3), 1997, pp. 241-258.
11. Manning, C.D., Raghavan, P., SchÃijtze, H., An Introduction to Information Retrieval, Draft, Cambridge University Press, 2009.
12. Melnik, S., Garcia-Molina, H.: Adaptive Algorithms for Set Containment Joins, ACM Transactions on Database Systems, Vol.28, No.2, 2003, pp.1-38.
13. Rivest, R.: Partial-Match Retrieval Algorithms. SIAM Journal on Computing 1976; 5(1).
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition, MIT Press, 2001.
15. Savnik, I., Flach, P.A.:, Bottom-up Induction of Functional Dependencies from Relations. Proc. of KDD'93 Workshop: Knowledge Discovery from Databases, AAAI Press, 1993, Washington, p. 174-185.
16. Savnik, I., Efficient subset and superset queries, Local Proceedings and Materials of Doctoral Consortium of the Tenth International Baltic Conference on Databases and Information Systems, 2012.
17. Savnik, I., Flach, P.A., Discovery of multivalued dependencies from relations, Intelligent Data Analysis Journal, Intelligent Data Analysis Journal, Vol.4, IOS Press, 2000, pp. 195-211.
18. Terrovitis, M., Passas, S., Vassiliadis, P., Sellis, T.: A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes, Proc. of ACM International Conference on Information and Knowledge Management, 2006.
19. Tousidou, E., Bozanis, P., Manolopoulos, Y.: Signature-based Structures for Objects with Set-valued Attributes, Information Systems 27, 2002, pp. 93-121.
20. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On Supporting Containment Queries in Relational Database Management Systems, ACM SIGMOD, 2001.