applied sciences

MDPI

Article

Ext-LOUDS: A Space Efficient Extended LOUDS Index for Superset Query

Lianyin Jia [1,2], Yuna Zhang [1], Jiaman Ding [1], Jinguo You [1], Yinong Chen [3] and Runxin Li [1,*]

1. Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650500, China; lianyinjia@kust.edu.cn (L.J.); NANA96307@hotmail.com (Y.Z.); jiamanding@kust.edu.cn (J.D.); jgyou@kust.edu.cn (J.Y.)
2. Yunnan Key Laboratory of Artificial Intelligence, Kunming University of Science and Technology, Kunming 650500, China
3. School of Computing, Informatics, and Decision Systems, Arizona State University, Tempe, AZ 85287, USA; yinong@asu.edu

* Correspondence: rxli@kmust.edu.cn

Received: 30 September 2020; Accepted: 24 November 2020; Published: 28 November 2020

Abstract: Superset query is widely used in object-oriented databases, data mining, and many other fields. Trie is an efficient index for superset query, whereas most existing trie index aim at improving query performance while ignoring storage overheads. To solve this problem, in this paper, we propose an efficient extended Level-Ordered Unary Degree Sequence (LOUDS) index: Ext-LOUDS. Ext-LOUDS expresses a trie by 1 integer vector and 3 bit vectors directly map each NodeID to its corresponding position, thus accelerating some key operations needed for superset query. Based on Ext-LOUDS, an efficient superset query algorithm, ELOUDS-Super, is designed. Experimental results on both real and synthetic datasets show that Ext-LOUDS can decrease 50%–60% space overheads compared with trie while maintaining a relative good query performance.

Keywords: Ext-LOUDS; ELOUDS-Super; LOUDS; trie

1. Introduction

With the rapid development in e-commerce, Internet of Things and many other fields, both the scale and complexity of data are increasing. Set is a powerful tool to express such kind of complex data. A collection of certain objects can be expressed by a set, such as commodities in a supermarket, students in a school, etc. With the increase of data scale, it is becoming a difficult task to retrieve a set in a large collection of sets.

Set query mainly includes set containment query and set similarity query. The former studies whether a set contains or is contained in another set, while the latter studies the containment degree between sets. Set containment queries include subset query, superset query, and equivalent query. In this paper, we focus on superset query, that is, given a query set Q, retrieve all subsets of Q in a set dataset D (Q is the superset of these sets).

Superset query is widely used in object-oriented database management systems (OODBMS) [1], job matching [2], data mining [3] and many other fields. Taking the object-oriented database management system as an example, in OODBMS, set values can be stored in attributes. A superset query algorithm can be deployed to quickly find all subsets of a given query set. For the job matching example, given a set of skills of a job seeker, a superset query can help him find all the jobs he can do.

Comparing the query with each set in a large collection is infeasible. Efficient indexes are key to improving superset query performance. Inverted index [4,5] is one of the most common indexes used in this field. Helmer et al. [6] designed an inverted index which scans all relevant inverted lists

and counts each set ID (SID) encountered. If the count of a SID equals to the number of elements in the corresponding set of the SID, then this set is a result of the superset query. Based on this idea, a lot of efforts have been put on inverted indexes. Terrovitis et al. [7] designed an index structure combining inverted index and B-tree to improve the performance on skewed distribution. Agrawal et al. [8] studied the problem of set containment query with error-tolerance.

The main drawbacks of inverted index based algorithms lies in that they require a large number of list scan operations, thus deteriorating the query performance. Unlike inverted index, trie can compress the common prefix of sets into a single path, thereby avoiding redundant access and improving query performance. In recent years, a lot of trie based efforts can be seen in set containment query. Jia et al. [9] designed an extended trie index called ETI by extending trie nodes with additional attributes to facilitate superset query. Based on this, an efficient superset query algorithm, E-Superset, is designed, which is efficient by only accessing a small number of nodes starting from the root. Although algorithms [2,10] in literature focus more on set containment join, their R-driven approach is essentially a superset query. In order to make full use of the compressed common prefixes, these algorithms usually build a partial or complete trie on dataset R, and then perform superset query for the records in dataset S to improve join efficiency. Yang et al. [5] combined trie with sampling technology to make a more accurate evaluation of the cardinality of containment query results.

Although trie can effectively improve the performance of superset query, it also introduces over-large space overheads [4]. For example, trie needs extra space to store the pointers pointing to the parent and child nodes of each node. In addition, to effectively support the set query, trie often needs to be extended with some attributes (e.g., the prefix set of current node [10], the link to the next node with the same label [9]) which usually are byte or integer types. These pointers and extensions will inevitably increase the overheads of trie, thereby affecting its scalability, especially when extended to large datasets.

Most trie-based superset query algorithms focus on query performance and ignore the storage space overheads. In response to this shortcoming, we try to research efficient trie compression technologies which can compress the space overheads and support superset queries. Double array [11,12] and Level-Ordered Unary Degree Sequence (LOUDS) [13–15] are the two most eminent compressed trie representatives. Compared with double array, LOUDS has a much higher compression ratio; however, its retrieving performance is relatively low [16,17]. Many works are devoting to solve this problem. Delpratt et al. [14] introduced double numbering which partitions LOUDS bit vector into 2 bit vectors according to runs of zeros and ones, then executes RANK & SELECT on shorter bit vector separately. Some recent works [18–20] researched efficient RANK & SELECT operations to improve retrieving performance of LOUDS. He et al. [21] designed a novel succinct structure that supports the mapping between preorder ranks and level-order ranks of nodes in constant time.

Differently, in this paper, we aim at exploiting LOUDS to facilitate efficient superset query. To do this, we design an efficient extended LOUDS index named Ext-LOUDS based on LOUDS. The core of Ext-LOUDS consists of 1 integer vector and 3 bit vectors. By developing efficient RANK and SELECT operations, the position of the parent node and child nodes of a certain node can be directly calculated. As a result, the pointer overheads can be easily avoided. In addition, extending trie nodes with bit vectors can further help us decrease the space overheads. Experimental results on two real datasets show that Ext-LOUDS can reduce space overheads by up to 50%–60% without significantly reducing query performance.

2. Problem Definition and Necessary Preliminaries

2.1. Set Superset Query

A database D consists of a collection of sets, where each set $S \in D$ comprises a set of elements drawn from a finite universe U. Each set S has a unique set identifier (SID). We assume that S is not

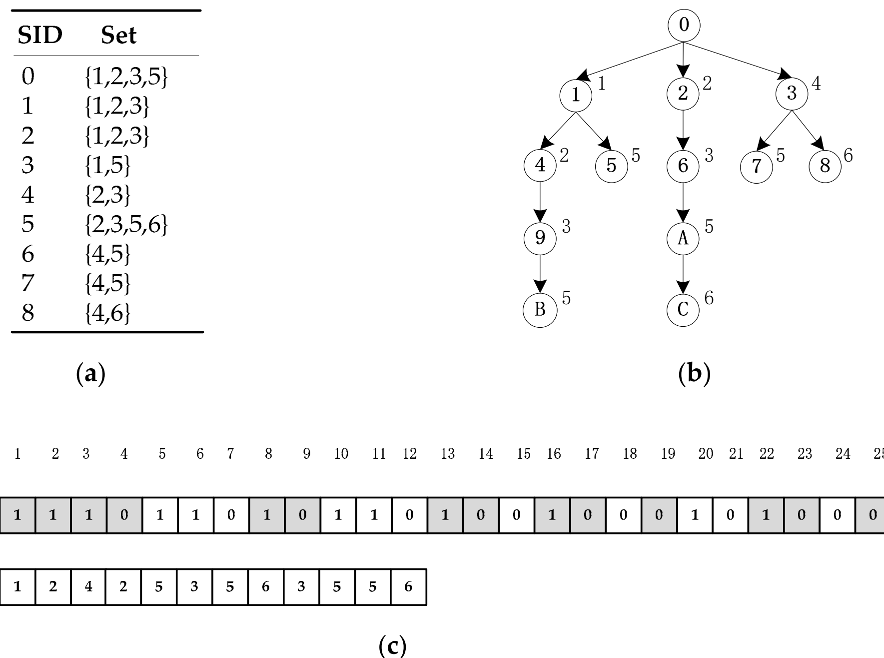
multiset, that is to say, there is no duplicate element in R. We use |D| to represent the number of sets in D, |U| to represent the number of distinct elements in D and |S| to represent the number of elements in *S*.

Now we formally define Set Superset Query as shown in Definition 1.

Definition 1. *(Set Superset Query). Given a query set Q and a database D, find all sets $S \in D$ satisfying $Q \supseteq S$.*

2.2. LOUDS

LOUDS encodes each trie node using a unary code, which has a compression ratio close to the theoretical lower bound of information theory. It is a compressed trie with the highest known compression ratio [17]. LOUDS has a simple encoding rule which encodes nodes hierarchically starting from the root node. When encoding a node, k bit 1s are set in a bit vector *B* for its k children and then another 0 is filled between this node and the next node. Elements are stored in vector Elems hierarchically. Given an example dataset in Figure 1a, the corresponding trie and LOUDS are shown in Figure 1b,c. The number to the right of each node in Figure 1b is the element of this node, and the number in each circle is its corresponding level ordered node ID (NodeID in short). The alternative background colors in LOUDS of Figure 1c denote the boundaries of each node.



SID	Set
0	{1,2,3,5}
1	{1,2,3}
2	{1,2,3}
3	{1,5}
4	{2,3}
5	{2,3,5,6}
6	{4,5}
7	{4,5}
8	{4,6}

Figure 1. An example dataset and its corresponding trie and LOUDS: (**a**) An example dataset; (**b**) trie constructed based on (**a**); (**c**) LOUDS.

Accessing the parent node and the child nodes in LOUDS can be realized by two operations: RANK and SELECT [22,23]. Given a position *p* (starting from 0) in LOUDS bit vector, $rank_b(p)$ returns the number of *b*s in range [0,*p*] and $Select_b(p)$ returns the position of the *p*-th *b* in LOUDS bit vector, where $b \in \{0,1\}$. In LOUDS, the position of the parent node and the first child node for a node starting from p can be computed by $Select_1(B, rank_0(B,p))$ and $Select_0(B, rank_1(B,p)) + 1$, respectively. The position of the *i*-th node can be obtained by $Select_0(B,i) + 1$.

3. Ext-LOUDS

3.1. Data Preprocessing

Data preprocessing maps or transforms data according to a specific logic, which can help to improve the index construction efficiency and superset query efficiency. Element ordering and set ordering are the two common preprocessing methods for sets. The former sorts elements in a set in a certain order, which further includes element value (EV: sort elements in a set by value) ordering, element frequency (EF: sort elements in a set by their frequency in the entire dataset) ordering and element frequency-value [9] (EFV: map elements to their positions in the frequency sequence) ordering. Set ordering sorts sets in a certain order, which includes set length (SL: sort sets according to their lengths) ordering, set dictionary (SD: view a collection of sets as strings and sort them alphabetically) ordering. For the example dataset in Figure 1a, the processed dataset by EFV and SD is shown in Figure 2.

SID	Set
0	{1,2}
1	{1,2,3,4}
2	{1,2,3,6}
3	{1,2,4}
4	{1,2,4}
5	{3,4}
6	{3,5}
7	{3,5}
8	{5,6}

Figure 2. Dataset processed by EFV and SD.

3.2. Ext-LOUDS

As mentioned earlier, LOUDS is a space-efficient compression trie representative, so we use LOUDS to compress trie in this paper. However, the simple LOUDS in Figure 1c cannot well support set superset query. The reasons are as follows: (1) LOUDS does not support duplicate sets widely existing in set datasets. (2) LOUDS cannot tell whether an internal node is an end node (A node corresponding to the last element of a set, e.g., NodeID 9 in Figure 1b). (3) LOUDS does not support storing the inverted lists on each node; (4) more importantly, some essential operations for superset query, e.g., checking whether a node has child, retrieving the first child of a node, are not efficient due to a long B in LOUDS and extra costs needed to map a position and its NodeID.

To address these issues, based on LOUDS, a new extended LOUDS named Ext-LOUDS is proposed. Ext-LOUDS organizes trie nodes hierarchically and avoids using pointers commonly used in trie, thus reducing space overheads.

The core of Ext-LOUDS is 1 integer vector and 3 bit vectors with a size of n (n is the number of trie nodes except root). To efficiently support superset queries while keeping a compact structure, we make necessary extensions on trie. Specifically, we model the 3 main node attributes of a trie (the first child of a node, whether a node has child nodes, whether a node is an end node) as 3 independent bit vectors with a length n to facilitate superset query, which makes checking these attributes much more efficient than using a single long bit vector B in LOUDS. Besides, in Ext-LOUDS, a position in each vector is also its corresponding NodeID, thus avoiding the additional computation operations to build mappings between them. As a result, the retrieving efficiency can be improved. Taking the dataset in Figure 2 as an example, the constructed trie and Ext-LOUDS are shown in Figure 3a,b, respectively.

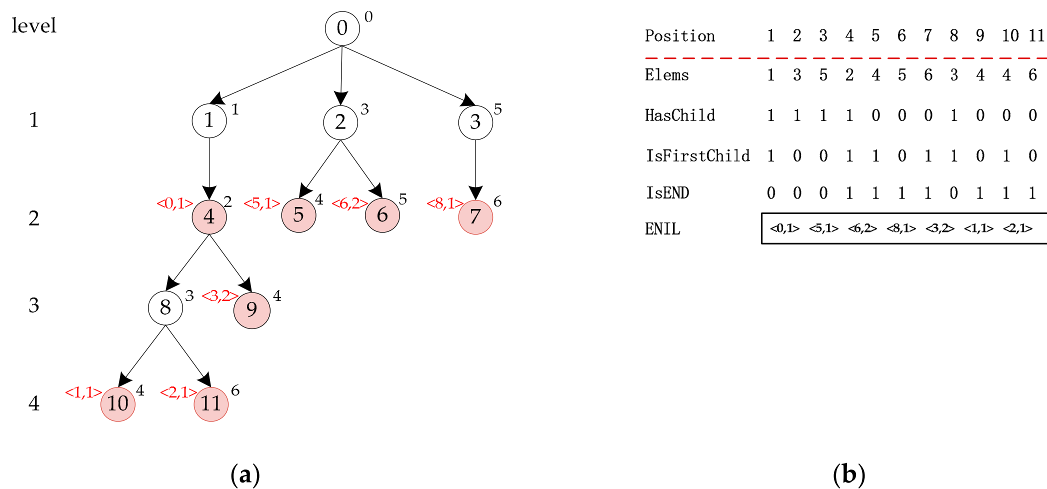


Figure 3. A trie and its corresponding Ext-LOUDS: (**a**) trie constructed according to Figure 2; (**b**) Ext-LOUDS.

The 4 vectors of Ext-LOUDS are introduced as follows:

The first vector, Elems, is an integer vector which sequentially records the element value of each node in a layer manner. We use an integer vector here because there are usually more independent elements in a set dataset than in a string dataset, so the commonly used character vector (supporting 256 independent elements) in LOUDS are not apt for sets.

The second vector is a bit vector, HasChild. A bit in HasChild indicates whether the corresponding node has child nodes. If it has, the bit is set to 1, 0 otherwise.

The third vector is a bit vector, IsFirstChild. A bit in IsFirstChild indicates whether the corresponding node is the first child node of its parent. If it is, the bit is set to 1, 0 otherwise.

The fourth vector is a bit vector, IsEND. A bit in IsEND indicates whether the corresponding node is an end node. If it is, the bit is set to 1, 0 otherwise. It should be noted that a leaf node must be an end node, but not vice versa.

In order to efficiently retrieve superset query results, in addition to the above 4 vectors, Ext-LOUDS also deploys a vector ENIL with a size of m (m is the number of end nodes in a trie) to store the inverted lists ending at each end node in a layer manner. Each element in ENIL is in a binary form of <startID, count> (shown on top left of the end nodes in Figure 3), where startID represents the start SID of the set corresponding to this end node, and count represents the number of duplicated sets corresponding to this end node. In addition, ENIL only needs to store m elements, instead of n elements in ordinary trie, which can further save space overheads.

It is important for Ext-LOUDS to support fast retrieval of parent and child nodes. Ext-LOUDS uses RANK and SELECT operations to achieve these goals as shown in Theorem 1 and Theorem 2, respectively.

Theorem 1. *Given an element e in Elems with a position p, the position of its first child node in Ext-LOUDS,* p_{child}*, can be calculated by* $p_{\text{child}} = select_1(IsFirstChild, rank_1(HasChild, p) + 1)$.

Proof. Assume $x = rank_1(HasChild, p)$, it means that there are $x + 1$ non-leaf nodes (including the root) corresponding to positions not greater than p in HasChild, as each non-leaf node has a first child, so the position of $(x + 1)$-th 1 in IsFirstChild gives the position of the first child of e. □

So Theorem 1 holds.

Theorem 2. *Given an element e in Elems with a position p, the position of e's parent,* p_{parent}*, can be calculated by* $p_{\text{parent}} = select_1(HasChild, rank_1(IsFirstChild, p) - 1)$.

Proof. Assume $y = rank_1(IsFirstChild, p)$, it means that there are y first child nodes corresponding to positions not greater than p in IsFirstChild, as each first child corresponds a non-leaf node, so the position of (y − 1)-th 1 in HasChild (not including the root) gives the position of the parent of e. □

So, Theorem 2 holds.

3.3. ELOUDS-Super Algorithm

Based on Ext-LOUDS, an efficient superset query algorithm, ELOUDS-Super, is proposed. The algorithm is implemented based on the following simple observation:

Observation 1. *Given a query Q, if all the elements corresponding to by the nodes from root (exclude root) to an end node N are in Q, the sets corresponding to N are qualifying results of Q.*

Based on the above observation, ELOUDS-Super works on a recursive manner and follows a depth-first traversal starting from the root. The detailed description of ELOUDS-Super is shown in Algorithm 1. In Algorithm 1, we use a parameter internalID to denote the sequence number of the current accessing internal node in the total internal nodes. At the beginning of the algorithm, both parameter level and internal ID are set to 0. The major steps of Algorithm 1 are discussed as follows:

1. Perform a SELECT operation on IsFirstChild vector to obtain the starting position p_{start} and the ending position p_{end} of the child nodes of node indicated by node_num (lines 2 and 3);
2. Perform a binary search to obtain the position p of the current query element $Q[level]$ in Elems vector (line 4);
3. If the node corresponding to p is an end node, perform a RANK operation on IsEnd vector to obtain the qualifying sets corresponding to the node, and merge it into the result set (lines 5–7);
4. If the node corresponding to p has child nodes, obtain the internalID and execute the algorithm recursively (lines 8–10).

Algorithm 1 ELOUDS-Super algorithm

//Input: *Q*, a query having the same ordering with dataset
Level: the depth of the trie, starts with 0
node_num: ID of non-leaf node, the ID of root is 0
//Output: *RS*: results of superset query
1. While($level$!=|Q|)
2. p_{start}←select$_1$($IsFirstChild$, $node_num$+1)
3. p_{end}←select$_1$($IsFirstChild$, $node_num$+2)
4. p←binary($Elems$, p_{start}, p_{end}, Q[$level$])
5. if($IsEnd$[p])
6. i←rank$_1$($IsEnd$, p)
7. RS←RS U $ENIL$[i]
8. if($Haschild$[p])
9. $node_num$←rank$_1$($HasChild$, p)
10. Superset(Q, $level$+1,$node_num$)
11. level←level+1

3.4. Algorithm Complexity Analysis

Space complexity: the core structure of Ext-LOUDS is an integer of length n, three bit vectors of length n, and a binary vector of length m. If the integer occupies four bytes, then the space overhead of Ext-LOUDS is (4 + 3/8) * n + 8 m bytes.

Time complexity: The time complexity of the algorithm is related to the number of visited nodes. In the worst case, the algorithm needs to visit $\frac{|Q|*(|Q|+1)}{2}$ nodes in total, so the time complexity of the algorithm is $O(|Q|_2)$.

4. Discussions

4.1. Experimental Environment and Datasets

In order to effectively evaluate the performance, we carried out extended experiments on both real datasets and synthetic datasets and compared Ext-LOUDS with inverted index, and trie. The experimental environment is shown below.

Hardware platform: Intel i7–7700 CPU @ 3.60GHz (4 cores and 8 hyper threads), 16 GB of memory. Software environment: Ubuntu 18.04 64-bit, Code: Blocks 16.01, gcc 7.5.0 as the default compiler.

4.1.1. Real Datasets

We use the following 2 real datasets. The first is MSWEB in the UCI KDD package [24]. This dataset is a week of access logs in the virtual area of Microsoft portal. Each record represents a set of areas accessed by a user session. MSWEB is a dense dataset, with a total of 32,711 records. The number of independent elements is 285 and the maximum length and minimum length are 35 and 1, respectively.

The second dataset is DBLP, which is a snapshot extracted from the famous DBLP bibliography [25]. The raw data is in XML format and we extract the author and editor field of the first 1 million publications to make a set dataset and remove the duplicate elements in each set. DBLP is a large and sparse dataset. The number of independent elements is 283,885, and the maximum length and minimum length are 176 and 1, respectively.

4.1.2. Synthetic Datasets

To evaluate the effects of the number of sets and the number of independent elements, we also generate zipf distributed synthetic datasets using similar settings with [6]. The parameters for generating the synthetic datasets are shown in Table 1.

Table 1. Dataset parameters.

Parameter	Symbol	Parameter Value
Dataset cardinality	$\|D\|$	50 k–300 k
Set cardinality	$\|R\|$	5–20
Distinct elements	$\|U\|$	1 k–50 k

Note that for both real and synthetic datasets, we use the dataset itself as a queries, then execute a self join and use the elapsed time to evaluate the query performance of all relevant algorithms.

4.2. Comparisons

4.2.1. Experiments on Real Datasets

The Effects of Different Ordering

In order to investigate the effect of different element ordering on index space overheads, the 2 datasets are processed by EFV and EV, respectively. The space overheads of Ext-LOUDS under different element orderings are examined, and the results of MSWEB and DBLP are shown in Figure 4a,b, respectively. It can be seen from these figures that EFV has a smaller space overhead compared with EV because more common prefixes can be compressed in EFV, thus leading to a fewer node number and a more compact structure. For MSWEB, the total number of nodes created by EFV is 23,819, 7.23% lesser than EV. For DBLP, the numbers are 2,945,469 and 12.18% respectively. This shows that EFV has a much higher space efficiency than EV. So, we use EFV as our default element ordering in the subsequent experiments.

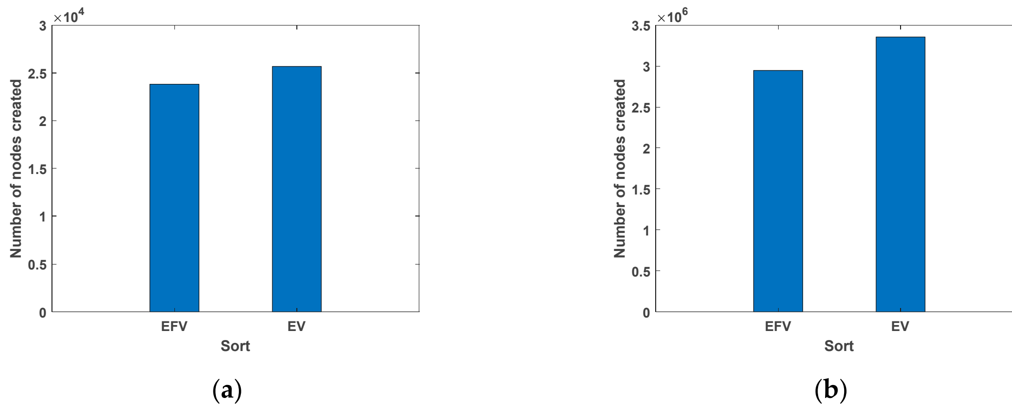


Figure 4. Space overheads for different element orderings: (**a**) MSWEB; (**b**) DBLP.

Compared with Different Indexes

In order to examine the space efficiency of different indexes, Ext-LOUDS is compared with inverted index and trie and the results for MSWEB and DBLP are shown in Figure 5a,b, respectively. It can be seen from these figures that the space overheads of Ext-LOUDS are much smaller than inverted index and trie. Compared with trie, the space overheads of Ext-LOUDS under MSWEB and DBLP are reduced 49.08% and 58.9%, respectively, which shows that Ext-LOUDS is much more space efficient than the other two. For MSWEB, trie consumes less space than inverted index because MSWEB is a small and dense dataset, so more common prefixes are compressed.

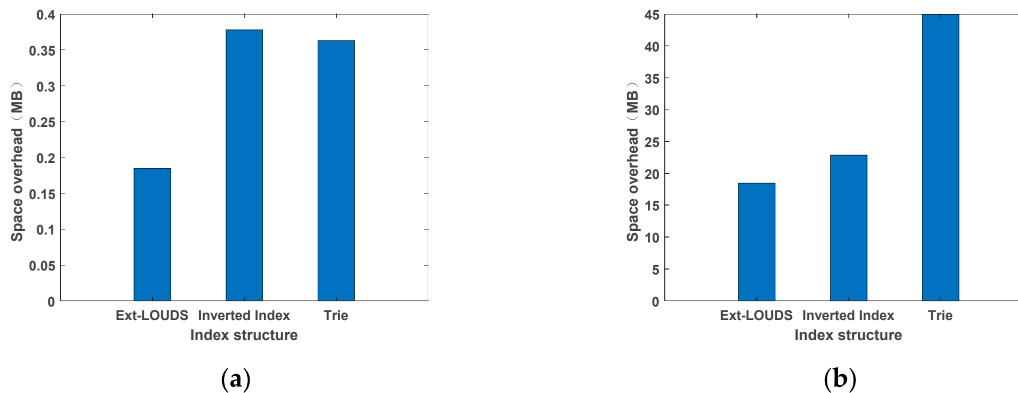
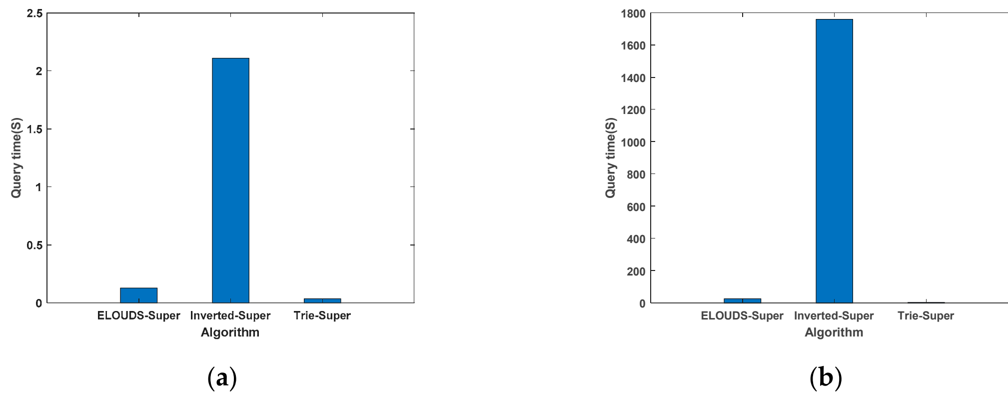


Figure 5. Space overheads for different indexes: (**a**) MSWEB; (**b**) DBLP.

The query efficiencies of the algorithms related to the 3 indexes are shown in Figure 6a,b, respectively. For ease of description, we call inverted index and trie based superset algorithm Inverted-Super and Trie-Super, accordingly. It can be seen from these figures that Inverted-Super is much slower than the other 2 algorithms. The reason is that Inverted-Super has to scan the related inverted lists, which causes a large time overheads. In contrast, Trie-Super and ELOUDS-Super only need to access a small number of nodes from the root, so a much higher query efficiency can be achieved. The query time of ELOUDS-Super is slightly higher than that of Trie-Super. The main reason is that ELOUDS-Super requires additional RANK and SELECT operations to retrieve the child nodes.

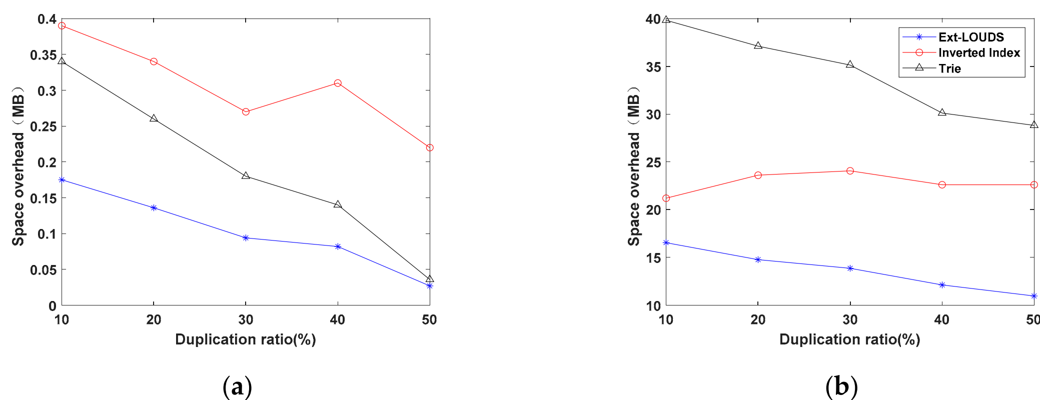


(**a**) (**b**)

Figure 6. Query times for the three algorithms: (**a**) MSWEB; (**b**) DBLP.

The Effects of Duplication Ratio

As duplicated sets are common in set datasets, we further study the effects of duplication ratio. To this end, we fix the total number of sets and randomly replace some sets with the other sets in the dataset to generate a new dataset with a specific duplication ratio (the proportion of duplicated sets to the total set). Figure 7a,b show the impact of duplication ratio varying from 10% to 50%. It can be seen from these figures that the space overheads of trie and Ext-LOUDS decrease apparently with the increase of the duplication ratio, while the duplication ratio has little impact on the inverted index. The reason lies in that the duplicated sets have a common path in trie, so the more duplicated records, the less space overheads. For inverted indexes, duplicated records will have independent SID in the related inverted lists, so the space overheads will not be changed significantly. In Figure 7a, the space overheads of inverted index vary with duplication ratio, which is because MSWEB is small and skewed. Therefore, the length deviation of the selected duplicated sets has a big impact on space overheads.



(**a**) (**b**)

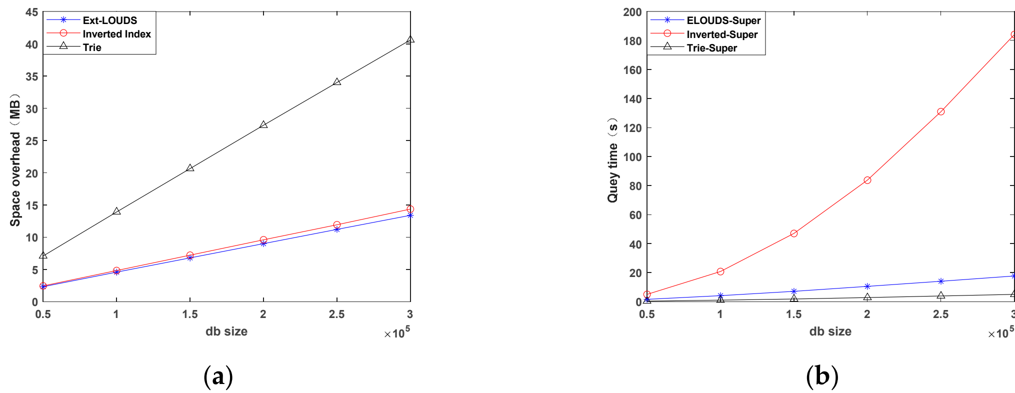
Figure 7. Space overheads for different duplication ratios: (**a**) MSWEB; (**b**) DBLP.

4.2.2. Experiments on Synthetic Datasets

Database parameters, such as the size of the dataset |D| and the number of distinct elements |U| in the dataset, also have a significant impact on the space and time overheads. We did a comparative experiment by fixing |D| and |U|, respectively.

The Effects of |D|

We carry out experiments on |U| fixed to 10,000 and |D| varied from 50,000 to 500,000. The space and time overheads are shown in Figure 8a,b, respectively. From these two figures, we can see that the space and time overheads increase with the increase of |D| for all the 3 indexes. Compared with the other 2 competitors, Ext-LOUDS has the best space overhead and has a query time overhead close to the best, which shows it has a good scalability and a good overall performance.

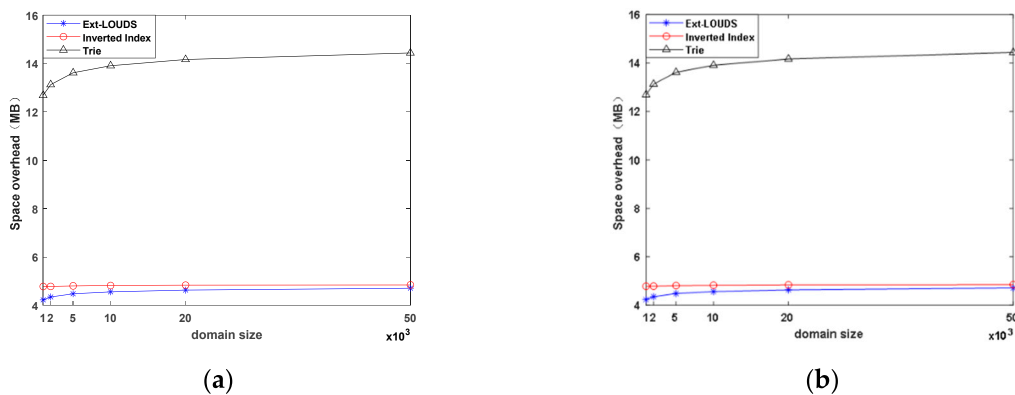


(**a**) (**b**)

Figure 8. Space and time overheads for varied |D|: (**a**) space overheads; (**b**) time overheads.

The Effects of |U|

We carry out experiments on |D| fixed to 100,000 and |U| varied from 1000 to 50,000. The space and time overheads are shown in Figure 9a,b, respectively. From these two figures, we can see that the space overheads for trie-based algorithms increase with the increase of |U|, whereas it has little impact on inverted indexes. The reason lies in that the smaller the |U| is, the more compact the trie will be. For an inverted index, the number of elements in all the inverted lists always equal to the total number of elements in the corresponding dataset, so its space overhead keeps stable. For the time overheads, the query time of all the 3 algorithms decrease with the increase of |U|. This is because a larger |U| means a shorter inverted list in inverted index or a lesser occurrence in trie that a query element may encounter, thus accelerating the query speed. Ext-LOUDS still has the best space overhead and has a query time overhead close to the best when compared with the other two indexes, which shows it has a good overall performance.



(**a**) (**b**)

Figure 9. Space and time overheads for varied |U|: (**a**) space overheads; (**b**) time overheads.

5. Conclusions and Future Work

In view of the high space overheads of trie, in this paper, an extended LOUDS structure, Ext-LOUDS, is proposed to efficiently support superset query. Compared to a long bit vector compressed in LOUDS, Ext-LOUDS compresses a trie into 4 shorter vectors. In Ext-LOUDS, a position is its NodeID, so some key operations needed for superset query are super fast. Based on Ext-LOUDS, an efficient superset query algorithm, ELOUDS-Super, is implemented. By extensive experiments on both real and synthetic datasets, we have the following findings: (1) Ext-LOUDS can significantly reduce the space overheads while maintaining a relative good query performance. (2) Ext-LOUDS works well on EFV and datasets with duplicated sets. (3) Ext-LOUDS scales well with varied |U| and |D|. As our future work, we plan to extend Ext-LOUDS to support other set containment query or similarity query. In addition, optimizing

the RANK and SELECT operations in Ext-LOUDS to further improve its query efficiency will also be our future work.

Author Contributions: Conceptualization, L.J. and R.L.; methodology, L.J. and Y.Z.; software, L.J. and Y.Z.; validation, L.J., Y.Z. and Y.C.; writing, L.J. and Y.Z.; resources, L.J., J.D. and J.Y; review, J.D., J.Y. and Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China under Grant no. 61562054, China Scholarship Council under Grant no. 201908530036, the National Key Research and Development Program of China under Grant no. 2018YFB1003904.

Conflicts of Interest: There are no conflicts of interest regarding the publication of this paper.

References

1. Savnik, I. Index Data Structure for Fast Subset and Superset Queries. In Proceedings of the International Conference on Availability, Reliability, and Security, Regensburg, Germany, 2–6 September 2013; pp. 134–148.
2. Deng, D.; Yang, C.; Shang, S. LCJoin: Set Containment Join via List Crosscutting. In Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE), Macau, China, 8–11 April 2019.
3. Bouros, P.; Mamoulis, N.; Ge, S.; Terrovitis, M. Set containment join revisited. *Knowl. Inf. Syst.* **2016**, *49*, 375–402. [CrossRef]
4. Luo, J.; Zhang, W.; Shi, S.; Gao, H.; Jiang, S. FreshJoin: An Efficient and Adaptive Algorithm for Set Containment Join. *Data Sci. Eng.* **2019**, *4*, 293–308. [CrossRef]
5. Yang, Y.; Zhang, W.; Zhang, Y.; Lin, X.; Wang, L. *Selectivity Estimation on Set Containment Search*; Springer: Cham, Switzerland, 2019; Volume 4.
6. Helmer, S.; Moerkotte, G. A performance study of four index structures for set-valued attributes of low cardinality. *Vldb J.* **2003**, *12*, 244–261. [CrossRef]
7. Terrovitis, M.; Bouros, P.; Vassiliadis, P.; Sellis, T.K.; Mamoulis, N. Efficient answering of set containment queries for skewed item distributions. In Proceedings of the International Conference on Edbt, Uppsala, Sweden, 21–24 March 2011.
8. Agrawal, P.; Arasu, A.; Kaushik, R. On indexing error-tolerant set containment. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, IN, USA, 6–10 June 2010.
9. Jia, L. *Research on Set Similarity and Set Containtment in Main Memory Database*; South China University of Technology: Guangzhou, China, 2012. (In Chinese)
10. Yang, J.; Zhang, W.; Yang, S.; Zhang, Y.; Lin, X. TT-Join: Efficient Set Containment Join. In Proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, USA, 19–22 April 2017.
11. Aoe, J.-I. An Efficient Digital Search Algorithm by Using a Double-Array Structure. *IEEE Trans. Softw. Eng.* **1989**, *15*, 1066–1077. [CrossRef]
12. Jia, L.; Zhang, C.; Li, M.; Chen, Y.; Ding, J. An Efficient Two-Level-Partitioning-Based Double Array and Its Parallelization. *Appl. Sci.* **2020**, *10*, 5266. [CrossRef]
13. Jacobson, G. Space-efficient static trees and graphs. In Proceedings of the Symposium on Foundations of Computer Science, Research Triangle Park, NC, USA, 30 October–1 November 1989.
14. Delpratt, O.N.; Rahman, N.; Raman, R. *Engineering the LOUDS Succinct Tree Representation*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 134–145.
15. Zhang, H.; Lim, H.; Leis, V.; Andersen, D.G.; Kaminsky, M.; Keeton, K.; Pavlo, A. Succinct Range Filters. *ACM Trans. Database Syst. (TODS)* **2020**, *45*, 1–31. [CrossRef]
16. Fuketa, M.; Kitagawa, H.; Ogawa, T.; Morita, K.; Aoe, J.I. Compression of double array structures for fixed length keywords. *Inf. Process. Manag.* **2014**, *50*, 796–806. [CrossRef]
17. Kanda, S.; Fuketa, M.; Morita, K.; Aoe, J.I. A compression method of double-array structures using linear functions. *Knowl. Inf. Syst.* **2016**, *48*, 55–80. [CrossRef]
18. Zhou, D.; Andersen, D.G.; Kaminsky, M. *Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 151–163.
19. Navarro, G.; Providel, E. *Fast, Small, Simple Rank/Select on Bitmaps*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 295–306.

20. Ermanno Pibiri, G.; Kanda, S. Rank/Select Queries over Mutable Bitmaps. *arXiv* **2020**, arXiv:2009.12809. Available online: https://arxiv.org/abs/2009.12809 (accessed on 10 November 2020).
21. He, M.; Munro, J.I.; Nekrich, Y.; Wild, S.; Wu, K. Distance Oracles for Interval Graphs via Breadth-First Rank/Select in Succinct Trees. *arXiv* **2020**, arXiv:2005.07644. Available online: https://arxiv.org/abs/2005.07644 (accessed on 13 November 2020).
22. Navarro, G.; Ord'On˜Ez, A. Grammar Compressed Sequences with Rank/Select Support. In *International Symposium on String Processing and Information Retrieval*; Springer: Cham, Switzerland, 2014.
23. Kai, B.; Johannes, F. High-Order Entropy Compressed Bit Vectors with Rank/Select. *Algorithms* **2014**, *7*, 608–620.
24. Bay, S.D.; Kibler, D.; Pazzani, M.J.; Smyth, P. The UCI KDD archive of large data sets for data mining research and experimentation. *SIGKDD Explor. Newsl.* **2000**, *2*, 81–85. [CrossRef]
25. Ley, M. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In Proceedings of the String Processing & Information Retrieval, International Symposium, Spire, Lisbon, Portugal, 11–13 September 2002.

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).