2010 12th International Asia-Pacific Web Conference

Efficient Common Items Extraction from Multiple Sorted Lists

Wei Lu [1,2], Chuitian Rong [1,2], Jinchuan Chen [2], Xiaoyong Du [1,2], Gabriel Pui Cheong Fung [3], Xiaofang Zhou [4]

[1] *School of Information, Renmin University of China* {uqwlu,rct682,jcchen,duyong}@ruc.edu.cn

[2] *Key Labs of Data Engineering and Knowledge Engineering, Ministry of Education, China*

[3] *School of Computing Informatics, Arizona State University* g.fung@asu.edu

[4] *School of ITEE, The University of Queensland, Australia* zxf@itee.uq.edu.au

***Abstract*—Given a set of lists, where items of each list are sorted by the ascending order of their values, the objective of this paper is to figure out the common items that appear in all of the lists efficiently. This problem is sometimes known as common items extraction from sorted lists. To solve this problem, one common approach is to scan all items of all lists sequentially in parallel until one of the lists is exhausted. However, we observe that if the overlap of items across all lists is not high, such sequential access approach can be significantly improved. In this paper, we propose two algorithms, MergeSkip and MergeESkip, to solve this problem by taking the idea of skipping as many items of lists as possible. As a result, a large number of comparisons among items can be saved, and hence the efficiency can be improved. We conduct extensive analysis of our proposed algorithms on one real dataset and two synthetic datasets with different data distributions. We report all our findings in this paper.**

I. Introduction

Given a set of sorted lists, where items of each list are sorted by the ascending order of their values, our objective is to figure out common items that appear in all of the sorted lists efficiently. This problem is sometimes known as *common items extraction from sorted lists*.

Common items extraction from sorted lists is a classical problem in computer science and has a wide range of applications in many different disciplines [1], [2], [3], [4], [5], [6], [7], such as:

- **Joins using sorted indexes** Assuming that we have n relations $R_1(X, Y_1)$, ..., $R_n(X, Y_n)$ with indexes on X for all relations, we want to compute $R_1(X, Y_1) \bowtie \ldots \bowtie R_n(X, Y_n)$. Notice that each item of the indexes is an X-value and indexes on X for all relations have been sorted. We then figure out the common items that appear in all indexes for join operation.
- **Information Retrieval** Assuming there exist a collection of documents and a query, our objective is to extract a set of documents, each of which contains all words in the query. In order to figure out the documents efficiently, we will always create a set of inverted lists first. Each inverted list represents the documents that contain a specific word. It usually contains a set of document ID's sorted in ascending order. Based on these inverted lists, we can obtain the required documents by figuring out the common ID's of those inverted lists that represent the words in the query.

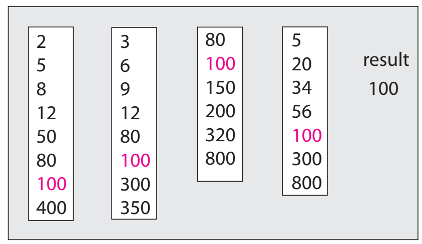


Fig. 1. An example

Example 1: Suppose there exist four ordered lists, shown in Figure 1. According to the algorithm described in [1], in order to obtain the common items of the lists, we start with the first items of the lists, which are 2, 3, 80, 5. If the current items are equal, then we know that the current item is one of the common items. Otherwise, we select the item with the minimum value and move to its next item. We repeat doing this judgement, selection and move until any of the lists is exhausted. Finally, the common item is 100.

As described in Example 1, each items of the lists needs to be accessed once before we finish scanning any of the lists. Moreover, comparisons among items of lists are required in order to obtain the items with the minimum value at each step. Actually, from Figure 1, we can observe that the maximum value of the first items of these lists is 80, from the third list. Clearly, items of the other lists, with values less than 80 cannot belong to the common items. As a result, we move to the first items of the other lists, with values greater than or equal to 80 instead of scanning items of lists one by one. Such strategy can be applied at each selection. As a result, performance can be of great improved if a large number of items are skipped such that comparisons among items of the lists can be saved.

To sum up, we make the following contributions:

- We propose two algorithms, MergeSkip and MergeESkip, to extract the common items from a set of sorted lists. For MergeSkip, it skips the items which are obvious not in the common items. For MergeESkip, we further improve

978-0-7695-4012-2/10 $26.00 © 2010 IEEE
DOI 10.1109/APWeb.2010.16

TABLE I
SYMBOLS AND THEIR DEFINITIONS

Symbol	Definition
$\mathcal{L}$	a set of sorted lists, $\mathcal{L} = \{L_1, \ldots, L_{\vert\mathcal{L}\vert}\}$
$\vert\mathcal{L}\vert$	the number of lists in $\mathcal{L}$
L_i	a sorted list, $1 \le i \le \vert\mathcal{L}\vert$
$\vert L_i\vert$	the number of items of L_i
$L_i[j]$	the item in position j of list L_i, $1 \le j \le \vert L_i\vert$

the performance of MergeSkip by skipping as many items of each list as possible;

- We conduct extensive experimental evaluation of our proposed algorithms on one real dataset and two synthetic datasets. The synthetic datasets are generated with different data distributions. The experimental results demonstrate the efficiency of our proposed algorithms.

II. PROBLEM DEFINITION

Let $\mathcal{L}$ be a set of lists. We use $|\mathcal{L}|$ to denote the number of lists and L_i to denote the i^{th} list in $\mathcal{L}$, where $1 \le i \le |\mathcal{L}|$. Given a list L_i, we use $L_i[j]$ to denote the item in position j of list L_i and $|L_i|$ to denote the number of items of L_i. An item can be an integer, a string or any other data structure provided that a total order relationship can be derived. For reference, Table I shows a set of symbols that will be used throughout this paper.

Definition 1 (Sorted List): Given a list L_i, L_i is said to be a sorted list if and only if: $\forall m, n, 1 \le m < n \le |L_i|$, $L_i[m] \le L_i[n]$.

For the ease of presentation, we assume that any list mentioned in this paper is a sorted list, formalized by Definition 1, and there is no duplication in any list. Dealing with non-ordered list is beyond the scope of this paper. Given a set of lists, our objective is to extract the common items that appear in all lists.

III. PROPOSED WORK

In this section, we first give a basic algorithm, MergeAll [1], to solve our problem. Then, we present our two proposed algorithms, MergeSkip and MergeESkip.

A. Existing Work: MergeAll

1) An Example: Assume there exist 4 lists shown in Figure 1. MergeAll begins by selecting the first items of four lists ($\mathcal{L} = \{L_1, L_2, L_3, L_4\}$), which are 2, 3, 80, 5. Since 2 is the item with the minimum value, we discard 2 of list L_1 and look at its next item, 5. Now, the current item of list L_2, 3, has the minimum value. As a result, we discard 3 of list L_2, and move to its next item, 6. Now, items of lists L_1 and L_4, have the minimum value, 5. Similarly, we discard items, 5's, of both L_1 and L_4, and move to the next items, 8 and 20, respectively. At each step, we choose items with the minimum value, and move to their next items. At the time when we reach the 7$^{\text{th}}$ of L_1, 6$^{\text{th}}$ of L_2, 2$^{\text{nd}}$ of L_3, 5$^{\text{th}}$ of L_4, we come across the common item, 100, and insert it to the result. Having dispensed with 100's, we move to items 400 of L_1, 300 of L_2, 150 of L_3, 300 of L_4. We repeat selecting items with minimum value, and moving to their next items. At the point when we reach items 400 of L_1, 350 of L_2, 800 of L_3, 800 of L_4, since 350 of L_2 has the minimum value, we move to its next item. However, L_2 is now exhausted, and we know there are no more items, belonging to the result.

Algorithm 1: $MergeAll(\mathcal{L})$

input : L: A set of lists
output: R: Common items of $\mathcal{L}$
1 initialize an array pos with size $|\mathcal{L}|$;
2 **for** $i \leftarrow 1$ **to** $|\mathcal{L}|$ **do**
3     $pos[i] \leftarrow 1$;
4 **while** *TRUE* **do**
5   **if** $L_1[pos[1]] = L_2[pos[2]] = \ldots = L_{\mathcal{L}}[pos[\mathcal{L}]]$ **then**
6         $R \leftarrow R \cup L_1[pos[1]]$;
7   **for** $i \leftarrow 1$ **to** $|\mathcal{L}|$ **do**
8   **if** $pos[i] = |L_i|$ **then return** R;
9             $pos[i] \leftarrow pos[i] + 1$;
10   **else**
11         $min \leftarrow \texttt{Min}\ (L_1[pos[1]], \ldots, L_{\mathcal{L}}[pos[\mathcal{L}]])$;
12   **foreach** $L_i \in \mathcal{L}$ **do**
13   **if** $L_i[pos[i]] = min$ **then**
14   **if** $pos[i] = |L_i|$ **then return** R;
15                 $pos[i] \leftarrow pos[i] + 1$;

2) Description of MergeAll: The details of MergeAll are described in Algorithm 1. We first initialize an array pos to store positions of current items (lines 1–3). If values of current items are equal (line 5), then we insert the item to the result set (line 6) and all of the lists will be advanced to the next item (lines 7–9). Notice that if any of lists is exhausted, then it is safe to return the result R (line 8). If values of the current items are not all equal, then we will identify items with the minimum value (line 11), and only those lists of which current items with values are equal to the minimum value will be advanced to the next items (line 12–15). The whole process continues until any of the lists is exhausted (lines 4–15).

In order to compare with our proposed algorithms later, Figure 2 shows the number of iterations and the number of scans using the dataset in Example 1. The algorithm terminates in the 19$^{\text{th}}$ iteration and the number of scanned items is 29.

B. Proposed Work 1: MergeSkip

In algorithm MergeAll, at each iteration, we choose items with the minimum value and move to their next items. We need to scan each list sequentially. Clearly, before any of the lists is exhausted for being scanned, items of each list are accessed one by one continuously. We cannot skip any of the items. Moreover, identifying the minimum key requires comparisons among items of all lists at each iteration. This is also a time consuming process. In order to improve the

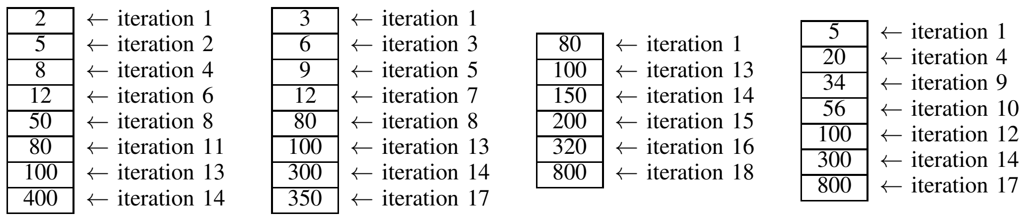


Fig. 2. Number of iterations and scans in Example 1 by using MergeAll. The algorithm terminates at Iteration 19 and 29 items are scanned.

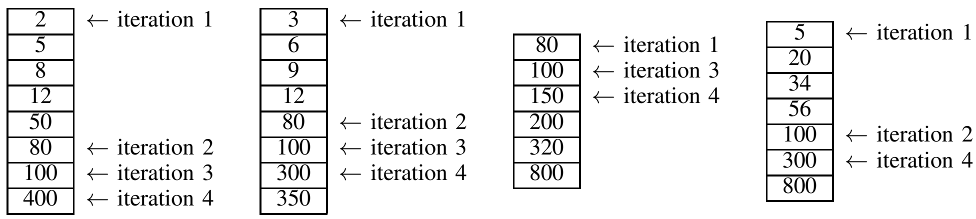


Fig. 3. Number of iterations and scans in Example 1 by using MergeSkip. The algorithm terminates at Iteration 5 and 14 items are scanned.

performance, we develop an algorithm MergeSkip based on the following observation:

Observation 1: Given any iteration of MergeAll, let minValue and maxValue be the minimum value and maximum value of current items of all lists, respectively. Then, an item with value greater than or equal to minValue and strictly smaller than maxValue cannot be a common item.

1) A Motivating Example: Let us consider the set of lists in Figure 1 again. At the first iteration, we move to the first items of each lists, which are 2, 3, 80 and 5 and figure out the item with the maximum value, 80, from list L_3. According to Observation 1, items with values less than 80 cannot be in the common items. At the second iteration, we can immediately move to the item 80 of L_1 (position 6), item 80 of L_2 (position 5) and item 100 of L_4 (position 5). As a result, ideally, we can skip 10 items. The whole process continues until any one of the lists is exhausted. Figure 3 shows the general idea of this algorithm. Ideally, the total number of iterations is only 5, and the total number of items we need to scan is only 14, which are both much fewer than these of MergeAll.

2) Description of MergeSkip: The details of MergeSkip are described in Algorithm 2. The main difference between MergeAll and MergeSkip is shown in lines 11 – 15. If values of the current items are not all equal, then we will calculate the maximum value (*maxValue*) among the current items of all lists (line 11). For each list L_i, if the value of its current item is not equal to *maxValue*, then we will issue a binary search from the current position $pos[i]$ to the end of list L_i, and identify the first item of L_i with its value greater than or equal to *maxValue* (lines 13–14). If values of all items of L_i are less than *maxValue*, then we stop the algorithm and immediately return R (line 15). The whole process continues until any one of the lists is exhausted (lines 4–15).

3) Discussion of MergeSkip: Comparing with MergeAll, MergeSkip offers a major advantage that items of the lists with their values less than *maxValue* are skipped at each iteration. This advantange is obvious if the overlap among all lists is low.

Algorithm 2: $MergeSkip(\mathcal{L})$

input : L: a set of lists
output: R: Common items in $\mathcal{L}$

```
1  initialize an array pos with size |L|;
2  for i ← 1 to |L| do
3      pos[i] ← 1 ;
4  while TRUE do
5      if L_1[pos[1]] = L_2[pos[2]] = ... = L_L[pos[L]] then
6          R ← R ∪ L_1[pos[1]];
7          for i ← 1 to |L| do
8              if pos[i] = |L_i| then return R;
9              pos[i] ← pos[i] + 1;
10     else
11         maxValue ← Max (L_1[pos[1]], ..., L_L[pos[L]]);
12         foreach L_i ∈ L do
13             if L_i[pos[i]] ≠ maxValue then
14                 biSearch (L_i, &pos[i], |L_i|, maxValue);
15                 if pos[i] = −1 then return R;
```

However, if all lists are very similar, using the binary search cannot derive too much benifit. In order to tackle this issue, we slightly modify the binary search based on the observation that in most cases, the item, which we try to search of each list L_i, is actually near the current position $pos[i]$. Thus, we try to shrink the search range which begins from $pos[i]$ to $|L_i|$. The modified binary search is conducted as follows:

We first check the item at position $pos[i]$ of L_i. If the value of the item is not less than $maxKey$, then this search of L_i is finished. Otherwise, we check the item at position $pos[i]+2$. If the value of the item is greater than $maxKey$, then we only check the item at position $pos[i]+1$. Otherwise, we check the item at position $pos[i]+4$. Generally, at the j^{th} ($j \geq 1$) step

, if we verify that the value of item at position $pos[i] + 2^j$ is less than $maxKey$, then we issue a binary search begining from $pos[i] + 2^{j-1} + 1$ to $pos[i] + 2^j - 1$. The performance can be improved as the search range is shrinked for the binary search.

C. Proposed Work 2: MergeESkip

In this part, we develop another enhanced version of MergeSkip, called MergeESkip, to solve our common items extraction problem. As described in MergeSkip, at each iteration, we select the item with the maximum value (*maxValue*) such that at the next iteration we can skip items of the other lists with their values less than *maxValue*. Notice that *maxValue* does not change throughout the *same* iteration, i.e. during the loop in lines 12 – 15 of Algorithm 2, *maxValue* is always the same for the same iteration. In contrast, in MergeESkip *maxValue* will be refined continuously even in the same iteration. This will be more effective and efficient if the distribution of values among the list is sparse.

1) A Motivating Example: Again, let us take an example, which is shown in Figure 4, using the lists in Figure 1 before we formally present the algorithm. We start by selecting the first item, 2, of list L_1 and set *maxValue* to 2. When we access list L_2, according to Observation 1, items with values less than 2 are skipped. Thus, we select the first item 3 of L_2, and update *maxValue* to 3. Similarly, we move to the first item 80 of L_3, and update *maxValue* to 80. When we access list L_4, we move to the item 100, which is the item at the 5th position, and update *maxValue* to 100. Hence, the first four items of L_4 are skipped. We continue the whole process until any one of the lists is exhausted. Eventually, we only need to access 8 items and there are totally 4 iterations involved.

2) Description of MergeESkip: The details of algorithm MergeESkip are described in Algorithm 3. We first set *maxValue* as the value of the first item of L_1 and set $counter$ to store how many current items of lists have the same value with *maxValue* (lines 1 – 3). In the manner similar to Algorithm 2, given a list L_i, we issue a binary search to obtain the item of L_i with the minimum value among all items, whose values are greater than or equal to *maxValue* (line 5). If any item of L_i has the value less than *maxValue*, then the algorithm terminates, and the result R is returned immediately (line 6). Otherwise, if the value of the current item of L_i is greater than *maxValue*, we update *maxValue* and reset $counter$ immediately (lines 7 – 9); otherwise, we increase the counter by 1 (line 13) so as to denote how many number of items has values equal to *maxValue*. If $counter$ is equal to the number of lists (line 14), this implies current items of all lists have the same values of *maxValue*. We therefore union *maxValue* into the result R, and reset necessary parameters in lines 11 – 17, which is self-explained. The whole process continues until any one of the lists is exhausted (lines 4 – 18).

3) Further Discussion of MergeESkip: In MergeESkip, taking different precedence of selecting lists can affect the performance. Taking Figure 4 for example, if we take the first item, 80, of L_3 to access first, then 2 of L_1, 3 of L_2 can be skipped, while these two items are accessed if we select L_1 and L_2 first. Thus, several strategies to take precedence of selecting lists can be adopted.

Algorithm 3: $MergeESkip(\mathcal{L})$

input : $\mathcal{L}$: a set of lists
output: R: Common Items in $\mathcal{L}$

1 initialize an array pos with the size $|\mathcal{L}|$;
2 $pos[1] \leftarrow 1$;
3 *maxValue* $\leftarrow L_1[pos[1]]$; *counter* $\leftarrow 1$; $i \leftarrow 2$;
4 **while** *TRUE* **do**
5   `biSearch` (L_i, &$pos[i]$, $|L_i|$, *maxValue*);
6   **if** $pos[i] = -1$ **then return** R ;
7   **if** $L_i[pos[i]] >$ *maxValue* **then**
8   *maxValue* $\leftarrow L_i[pos[i]]$;
9   *counter* $\leftarrow 1$;
10   **else**
11   *counter* $\leftarrow$ *counter* $+ 1$;
12   **if** *counter* $= |\mathcal{L}|$ **then**
13             $R \leftarrow R \cup$ *maxValue*;
14   **if** $pos[i] = |L_i|$ **then return** R ;
15             $pos[i] \leftarrow pos[i] + 1$;
16   *maxValue* $\leftarrow L_i[pos[i]]$;
17   *counter* $\leftarrow 1$;
18     $i \leftarrow i = \mathcal{L}\ ?\ 1 : i + 1$;

- Selection in a Token Ring Method: As described in the previous part, we always take the next list to refine $maxValue$. If we reach the last list, then the first list is taken as the next list;
- Random Selection: We randomly select list as the next list to refine $maxValue$;
- Selection by Size of List: We select the list, which has the minimum length beginning from the current position to the end of the list.
- Selection by Statistical Information: We select the next list according to some statistical information of the list, which can be updated during the access.

In this paper, we apply strategy 1 to algorithm MergeESkip. Studying other strategies will be regarded as our future work.

IV. Experimental Evaluation

A. Experiment Setup

All experiments are conducted on a PC with Intel 2.0GHz dual core CPU and 1GB memory under Ubuntu 8.04.2 operating system. All programs are implemented in C as in-memory algorithms, with all lists loaded into memory before they are run, and compiled using GCC 4.2.4. We use the following commonly used real data set and two synthetic data sets. They cover a wide range of distributions and application domains.

- DBLP: DBLP is a computer science bibliography website[1], where each article is taken as a record, including

[1] http://www.informatik.uni-trier.de/∼ley/db/

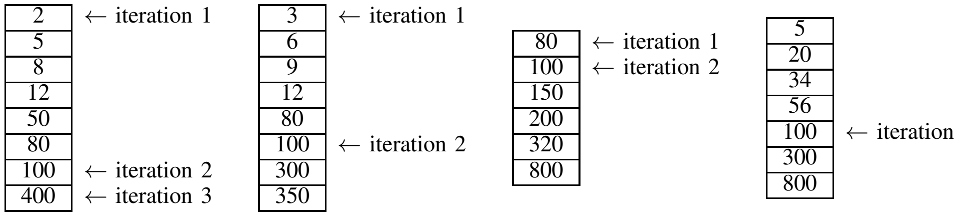
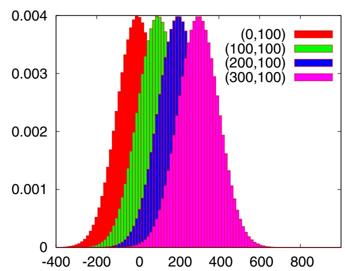


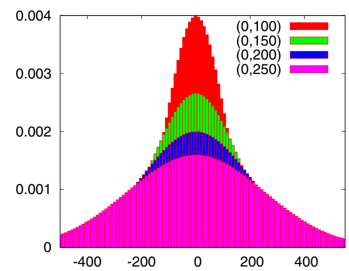
Fig. 4. Number of iterations and scans in Example 1 using MergeESkip. The algorithm terminates at Iteration 3 after scanning List 2 and 8 items are scanned.

list	Max	Min	Mean	S.D
1	769175	18	389810	221664
2	769179	70	400000	223967
3	769183	43	400200	221634
4	769172	1	388000	221411
5	769174	4	394000	222789
6	769172	14	388000	222294
7	769183	20	400200	222852
8	769178	14	386000	222236
9	769183	7	390000	223683
10	769174	5	400200	223047

(a) DBLP

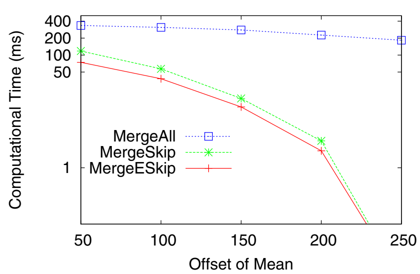


(b) different mean, the same variance

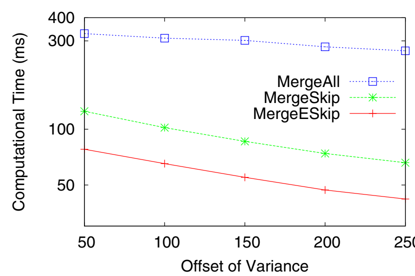


(c) the same mean, different variance

Fig. 5. Distribution of Different Data Sets



(a) different mean, the same variance



(b) the same mean, different variance

Fig. 6. Computational Time of Different Data Distributions

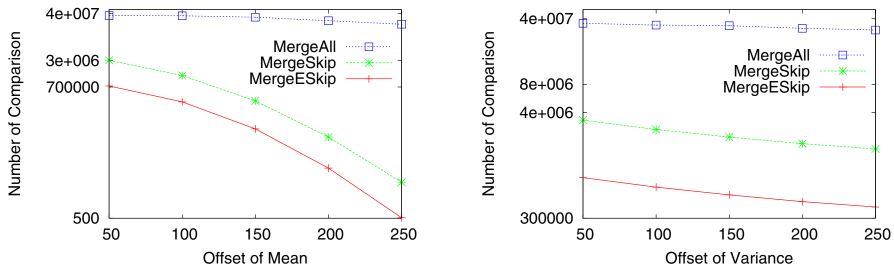
authors' names, the title, and so on. In this paper, we pre-construct the inverted index for the author' name, and randomly select ten inverted lists to figure out the common items extraction. The statistic information of such inverted lists is shown in Figure 5(a);

- Normal distribution with the same variance, but different mean: We generate a set of lists, where items of each list follow the normal distribution. For the first list, we set parameters, mean (0) and variance (100). For any other list, we set parameters, the same variance, but the mean is equal to that of previous list plus an offset. We control the overlap between these lists by the value: offset. An example of this data set is shown in Figure 5(b);
- Normal distribution with the same mean, but different variance: We generate a set of lists, where items of each list follow normal distribution. For the first list, we set parameters, mean (0) and variance (100). For any other list, we set parameters, the same mean but the variance is equal to that of previous list plus an offset. We control the overlap between these lists by the value: offset. An example of this data set is shown in Figure 5(c).

B. Effect of Data Distribution

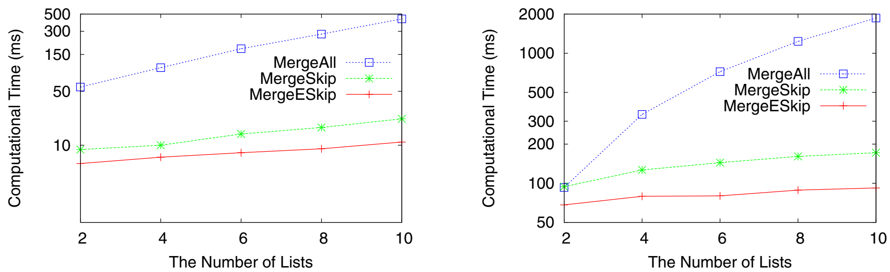
We investigate the performance of identifying the common items extraction problem on the two synthetic data sets, where $|\mathcal{L}|$=4 and $\forall L_i \in \mathcal{L}$, $|L_i|$=1M, and plot the results in Figure 6. We initialize the mean to 0, variance to 100, and set an offset in {50,100,150,200,250} for both two data sets.

From Figure 6, we can observe that MergeESkip performs the best, followed by MergeSkip, and MergeAll. To be specific, MergeESkip runs about 1.5 times faster than MergeSkip, and MergeSkip runs 3–4 times faster than MergeAll. Comparing with MergeSkip, MergeESkip reaches a item in a further position of the list such that more number of items can be skipped. Comparing with MergeAll, which scans items of lists one by one continuously before any list is exhausted, MergeSkip skips items, which cannot obvious be in the common items. When the value of offset increases, which means that the overlap across lists becomes smaller, all three algorithms run much faster. The reason that the efficiency of MergeAll improves is that some list finishes being scanned at a earlier time when the overlap decreases (notice that the size of all lists is 1M). For both MergeSkip and MergeSkip, when



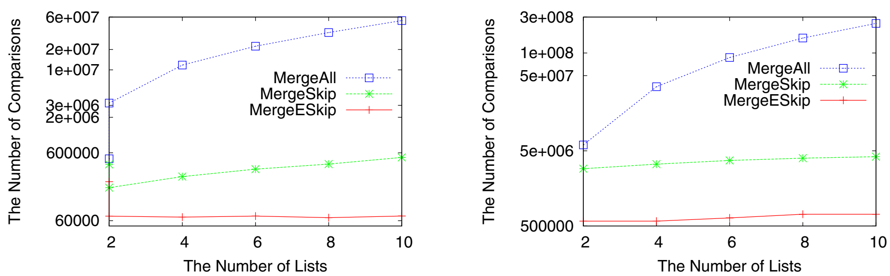
(a) different mean, the same variance (b) the same mean, different variance

Fig. 7. Number of Comparison of Different Data Distributions



(a) DBLP (b) the same mean, different variance

Fig. 8. Computational Time by Effect of Number of Lists



(a) DBLP (b) different mean, the same variance

Fig. 9. The Number of Comparisons by Effect of Number of Lists

the overlap decreases, the ability of skip becomes much more powerful.

In order to further illustrate the advantage that this skip strategy brings, we count the number of comparisons in each algorithm, which is shown in Figure 7. We can observe that, basically, no matter what the distribution is, the trend of computational time is in accordance with the trend of the number of comparisons, which demonstrates the computational time is dependent on the number of comparisons.

C. Effect of Number of Lists

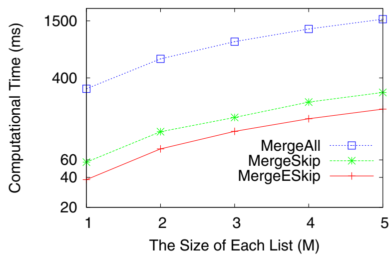
We study the performance of identifying the common items extraction problem on the real data set and the synthetic data set 2, where $|\mathcal{L}|$ in {2,4,6,8,10} and $\forall L_i \in \mathcal{L}$, $|L_i|$=1M, and plot the results in Figure 8. In the synthetic data set, we initialize the mean to 0, variance to 100, and offset to 100.

Still, MergeESkip performs the best, followed by MergeSkip and MergeAll. When the number of lists increases, the computational time of the algorithms increases linearly, and the gaps between MergeAll and MergeSkip, MergeSkip and MergeESkip become larger. The reason for MergeAll is that we need to access items of the new added lists. For MergeSkip and MergeAll, although new lists are added, more number of items of old lists can be skipped, since $maxValue$ may be refined in the new added lists, and items of new lists can be skipped as well. We also count the number of comparisons in each algorithm, which is shown in Figure 9. We can observe that, the trend of computational time is in accordance with that of the number of comparisons.

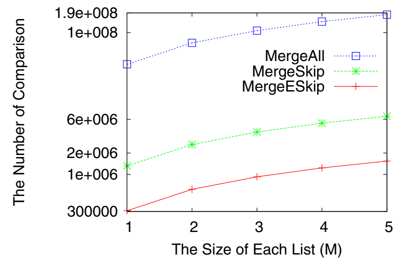
D. Effect of Size of Lists

We measure the performance of identifying the common items extraction problem on the synthetic data set 1, where $|\mathcal{L}|$ = 4 and $\forall L_i \in \mathcal{L}$, $|L_i|$ in {1M, 2M, 3M, 4M,5M}, and plot the results in Figure 10(a). In synthetic data set 1, we initialize the mean to 0, variance to 100, and offset to 100.

From the figure, we can observe that MergeESkip performs the best, followed by MergeSkip and MergeAll. When the



(a) Computational Time



(b) The Number of Comparisons

Fig. 10. Effect of Size of Lists

size of each list increases, the computational time of the algorithms increases linearly. However, when the size of each list increases, the gaps between MergeAll and MergeSkip, MergeSkip and MergeESkip keep the same. This finding shows that when the size of the list increases, the power of skip keeps nearly the same for the same data distribution. We also count the number of comparisons in each algorithm, which is shown in Figure 10(b). We can observe that, the trend of computational time is in accordance with that of the number of comparisons as well.

V. Related Work

Common items extraction problem is widely applied not only in the database community, but also in many other communities of the computer science.

In database community, the common items extraction problem has been studied in [1], [8], [9], [10]. In [1], they propose an algorithm to perform join operation on two ordered lists (it can also be extended and applied to multiple ordered lists, which is described by MergeAll algorithm). Our proposed algorithms are based on their proposed algorithm and are more efficient because of the skip technique. In [8], [10], [9], they focus on the problem of how to efficiently figure out items, appearing in a given number of lists, rather than all the lists, which has different motivations with ours.

Common items extraction problem is also studied in [11], [12], [13]. In [11], [12], Hollaar studies how to construct a specialized processor to speed up the calculation of common items from the instruction level of CPU. In [13], Stellhorn describes a specialized computer system, which is capable of performing common items extraction, which takes information retrieval as application scenario, in hardware. This architecture can derive significant improvement of performance. Different from the previous work, we focus on the efficiency of the common items extraction problem by decreasing the number of comparisons as many as possible.

VI. Conclusion

In this paper, we study the problem of identifying a set of common items that appear in all of the sorted lists efficiently. Existing work, which handles this problem, needs to scan the items of all lists until any one of the lists is exhausted for being scanned. In this paper, we propose two algorithms, MergeSkip and MergeESkip, that try to solve this problem by taking the idea of skipping as many items of lists as possible. Based on this skip strategy, a large number of comparisons among items of lists can be saved, and the efficiency is improved. We conduct a set of comprehensive experiments on a real dataset and two large-scale synthetic datasets. The results demonstrate that our proposed approaches are effective and efficient especially when the overlap across all lists is not high.

VII. Acknowledgements

This research is partially supported by the National Natural Science Foundation of China under Grant No.60873017, Graduate Research Fund Project of Renmin University of China under Grant No.10XNH097.

References

[1] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database System Implementation*. Prentice-Hall, 2000.
[2] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.
[3] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: sextuple indexing for semantic web data management,” *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.
[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O’Neil, P. E. O’Neil, A. Rasin, N. Tran, and S. B. Zdonik, “C-store: A column-oriented dbms,” in *VLDB*, 2005, pp. 553–564.
[5] A. L. Holloway and D. J. DeWitt, “Read-optimized databases, in depth,” *PVLDB*, vol. 1, no. 1, pp. 502–513, 2008.
[6] D. J. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in *SIGMOD Conference*, 2006, pp. 671–682.
[7] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, “Structural joins: A primitive for efficient xml query pattern matching,” in *ICDE*, 2002, pp. 141–.
[8] Y. L. Chen Li, Jiaheng Lu, “Efficient merging and filtering algorithms for approximate string searches,” in *ICDE*, 2008, pp. 257–266.
[9] C. Li, B. Wang, and X. Yang, “Vgram: Improving performance of approximate queries on string collections using variable-length grams,” in *VLDB*, 2007, pp. 303–314.
[10] S. Sarawagi and A. Kirpal, “Efficient set joins on similarity predicates,” in *SIGMOD*, 2004, pp. 743–754.
[11] L. A. Hollaar, “An architecture for the efficient combining of linearly ordered lists,” *SIGIR Forum*, vol. 10, no. 4, pp. 16–21, 1976.
[12] L. Hollaar, “A list merging processor for inverted file information retrieval systems,” in *University of Illinois at Urbana-Champaign*, 1975.
[13] W. Stellhorn, “An inverted file processor for information retrieval,” *IEEE Transactions on Computers*, vol. 26, pp. 1258–1267, 1977.