Overlap Set Similarity Joins with Theoretical Guarantees

Dong Deng
MIT CSAIL
dongdeng@csail.mit.edu

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

ABSTRACT

This paper studies the *set similarity join problem with overlap constraints* which, given two collections of sets and a constant c, finds all the set pairs in the datasets that share at least c common elements. This is a fundamental operation in many fields, such as information retrieval, data mining, and machine learning. The time complexity of all existing methods is $O(n^2)$ where n is the total size of all the sets. In this paper, we present a size-aware algorithm with the time complexity of $O(n^{2-\frac{1}{c}}k^{\frac{1}{2c}}) = o(n^2) + O(k)$, where k is the number of results. The size-aware algorithm divides all the sets into small and large ones based on their sizes and processes them separately. We can use existing methods to process the large sets and focus on the small sets in this paper. We develop several optimization heuristics for the small sets to improve the practical performance significantly. As the size boundary between the small sets and the large sets is crucial to the efficiency, we propose an effective size boundary selection algorithm to judiciously choose an appropriate size boundary, which works very well in practice. Experimental results on real-world datasets show that our methods achieve high performance and outperform the state-of-the-art approaches by up to an order of magnitude.

CCS CONCEPTS

• **Information systems → Join algorithms**; **Information integration**;

KEYWORDS

Similarity Join; Overlap; Set; Scalable; Sub-quadratic; Theoretical Guarantee

ACM Reference format:
Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD'18),* 16 pages.
https://doi.org/http://dx.doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Set similarity join with overlap constraints, which, given two collections of sets (e.g., the topic set of a document) and a constant c, finds all the set pairs that share at least c common elements, is a fundamental operation in many applications, such as word embedding [23], recommender systems [35], and matrix factorization [29]. Given a collection of documents, where each document contains a bag of words, for each word we can get a set of documents containing this word. Many prestige word embedding models, such as the deep learning based models `GloVe` [23] and `Word2Vec` [20] and the classical matrix factorization based models `HAL` (Hyperspace Analogue to Language) [16] and `PPMI` (Positive Pointwise Mutual Information) [6], make use of the number of documents in which a pair of words co-occurs. The overlap set similarity join can build the word co-occurrence matrix for these models, as the co-occurrence of two words is the same as the overlap size of their corresponding document sets. In addition, the recommender systems [35] often use the overlap (e.g., "*sharing c common friends*" in Facebook) to explain the recommendations for better transparency and user experience.

In terms of algorithm design, this problem is interesting from the perspectives of both theory and practice. Theoretically speaking, it admits a naive solution that simply compares all pairs of sets, and finishes in $O(n^2)$ time, where n is the total size of all the sets. Existing approaches [3, 13, 30] utilize various heuristics to improve efficiency. However, unfortunately, all of them are still captured by the $O(n^2)$ bound, namely, asymptotically as bad as the naive solution. Practically speaking, it has been observed that the "realistic" inputs to the problem appear much easier than the theoretical "worst case", which explains the community's enthusiasm for purely heuristic solutions so far.

This paper makes progress on both fronts simultaneously. At the philosophical level, we show that there does not need to be a fine line between theory and practice, as opposed to what was conceived previously. For this purpose, we propose a framework that (i) in theory, gives the first algorithm that escapes the quadratic trap, and (ii) in practice, can be easily integrated with clever heuristics to yield new solutions that improve the efficiency of the state-of-the-art. Specifically, our contributions can be summarized as follows.

Theoretical Guarantee: We present a size-aware algorithm that has the time complexity of $O(n^{2-\frac{1}{c}}k^{\frac{1}{2c}}) = o(n^2) + O(k)$, where k is the number of results. This is $o(n^2)$ as long as $k = o(n^2)$; on the other hand, if $k = \Omega(n^2)$, then any algorithm must incur $\Omega(n^2)$ time just to output the results. Therefore, our algorithm beats the quadratic complexity, whenever possible.

Practical Performance: The size-aware algorithm divides all the sets into small sets and large sets based on their sizes and processes them separately using two different methods. The two methods are size sensitive, i.e., one method is more efficient for small sets and the other one is more effective for large sets. We can utilize existing studies to process large sets, and focus on the small sets in this paper. For the small sets, we enumerate all their subsets with size c and take any two small sets sharing a common subset as a result. We develop optimization techniques to avoid enumerating a

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/XXXXXX.XXXXXX

	id	set	size
$\mathcal{R}_s$	R_1	$\{e_1, e_2, e_3\}$	3
	R_2	$\{e_1, e_3, e_4, e_7\}$	4
	R_3	$\{e_1, e_3, e_5, e_7\}$	4
	R_4	$\{e_2, e_4, e_5, e_6\}$	4
$\mathcal{R}_l$	R_5	$\{e_2, e_4, e_5, e_6, e_8, e_9, e_{10}, e_{11}\}$	8
	R_6	$\{e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}\}$	8
	R_7	$\{e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}\}$	9

Table 1: A collection $\mathcal{R}$ of sets

huge number of unnecessary subsets and improve the practical performance dramatically. Furthermore, as the size boundary between the small sets and the large sets is crucial to the efficiency, we propose an effective size boundary selection algorithm to judiciously choose a size boundary. Our optimization techniques can improve the practical performance dramatically. We have conducted extensive experiments on real datasets and the experimental results show that our method outperforms state-of-the-art methods by up to an order of magnitude.

The rest of the paper is organized as follows. We formulate the problem in Section 2. Section 3 introduces the size-aware algorithm. We develop the optimization heuristics for the small sets in Section 4. We propose the size boundary selection algorithm in Section 5. Section 6 reports the experimental results. We review related work in Section 7 and conclude in Section 8.

2 PROBLEM DEFINITION

Given two collections of sets, the set similarity join problem aims to find all the *similar* set pairs from the two collections. We use the overlap similarity to measure the similarity between two sets in this paper. Given two sets R and S, their overlap similarity is their intersection size, namely $|R \cap S|$. Two sets are said to be *similar* if and only if their overlap similarity is at least a given threshold c, i.e., $|R \cap S| \geq c$. Next we formally define the problem of set similarity joins with overlap constraints.

Definition 1. *Given two collections of sets $\mathcal{R}$ and $\mathcal{S}$ and a constant c, the set similarity join with overlap constraints reports all the set pairs $\langle R, S\rangle \in \mathcal{R} \times \mathcal{S}$ such that $|R \cap S| \geq c$.*

We first focus on the self-join case in this paper, i.e., $\mathcal{R} = \mathcal{S}$. Our technical and theoretical results can be seamlessly extended to the case of $\mathcal{R} \neq \mathcal{S}$, which are discussed in Appendix B. For example, consider the dataset $\mathcal{R}$ in Table 1. Suppose the overlap similarity threshold c is 2. R_1 and R_2 make a similar set pair as $|R_1 \cap R_2| = 2 \geq c$. In our running example, $\langle R_1, R_3\rangle$, $\langle R_2, R_3\rangle$, $\langle R_4, R_5\rangle$ and $\langle R_6, R_7\rangle$ are also similar pairs.

A brute-force method enumerates every set pair in $\mathcal{R} \times \mathcal{R}$ and calculates their overlap size. Let $n = \sum_{R_i \in \mathcal{R}} |R_i|$ be the total size of all sets. The brute-force method has a time complexity of $O(n^2)$.

3 A SIZE-AWARE ALGORITHM

In this section, we present an algorithm that solves the set similarity join problem with running time $o(n^2) + O(k)$, where k is the number of pairs in the result. This is the first algorithm that beats the quadratic time complexity of this problem whenever it is possible. Section 3.1 will describe the overall framework of our solution, but leave open the choice of a crucial parameter. Section 3.2 will explain how to set that parameter to achieve the best time complexity. We will discuss how to improve the practical performance of this algorithm in Sections 4 and 5.

Algorithm 1: SizeAwareAlgorithm

Input: $\mathcal{R}$: the dataset $\{R_1, R_2, \ldots, R_m\}$; c: threshold;
Output: $\mathcal{A} = \{\langle R_i, R_j\rangle \big| |R_i \cap R_j| \geq c\}$;

```
1  x = GetSizeBoundary(R, c);
2  divide R into small sets R_s and large sets R_l by x;
3  foreach large set R_i ∈ R_l do
4      foreach R_j ∈ R do
5          if |R_i ∩ R_j| ≥ c then insert ⟨R_i, R_j⟩ into A
6  foreach small set R_j ∈ R_s do
7      foreach c-subset r_c of R_j do
8          append R_j to L[r_c];
9  foreach inverted list L[r_c] in L do
10     add every set pair in L[r_c] into A;
11 return A;
```

3.1 The Framework

We will use the term *c-subset* to refer to any set of c elements (drawn from the sets in $\mathcal{R}$ and $\mathcal{S}$). It is easy to see that, two sets are similar if and only if they share a common c-subset. The observation motivates us to build an inverted index on all the c-subsets to aggregate those sets sharing common c-subsets, and compute the join result by examining each inverted list in turn. Thus, we avoid the enumeration of dissimilar set pairs, i.e., set pairs that do not share any common c-subsets. This approach, however, works well only for sets with small sizes, as they have a small number of c-subsets. On the other hand, the number of large sets cannot be very large, such that we can afford to apply even a "brute-force" method on them. Next, we develop these ideas into a formal algorithm.

Given a collection $\mathcal{R}$ of sets $R_1, R_2, \ldots, R_m$ and an overlap similarity threshold c, we divide all the sets into two categories based on their sizes. The first category $\mathcal{R}_l$ contains all the sets with sizes at least x—the selection of the size boundary x will be discussed later—which we refer to as the *large sets*. The second category $\mathcal{R}_s$ contains all the sets with sizes smaller than x, which we refer to as the *small sets*. Obviously, any similar set pair in $\mathcal{R} \times \mathcal{R}$ can be found in either $\mathcal{R}_l \times \mathcal{R}$ or $\mathcal{R}_s \times \mathcal{R}_s$.

We obtain the similar set pairs in $\mathcal{R}_l \times \mathcal{R}$ and $\mathcal{R}_s \times \mathcal{R}_s$ in different ways:

- For $\mathcal{R}_l \times \mathcal{R}$, simply enumerate every set pair in $\mathcal{R}_l \times \mathcal{R}$, and calculate their intersection size.
- To find all the similar set pairs from $\mathcal{R}_s \times \mathcal{R}_s$, we first build a c-subset inverted index $\mathcal{L}$ for all the c-subsets in the small sets. The inverted list $\mathcal{L}[\mathtt{r}_c]$ consists of all the small sets that contain the c-subset $\mathtt{r}_c$. Then, we access each inverted list, and add every set pair in it into the result set (i.e., for any two distinct sets R and R' in the inverted list, add $\langle R, R'\rangle$ to the result). This produces all the similar set pairs in $\mathcal{R}_s \times \mathcal{R}_s$.

The pseudo-code of the above algorithm is shown in Algorithm 1. It takes a collection of sets $\mathcal{R}=\{R_1, R_2, \ldots, R_m\}$ and a constant threshold c as input, and outputs all the similar set pairs. It first calculates the size boundary x, and then divides all the sets in $\mathcal{R}$ into two categories, the small sets $\mathcal{R}_s$ and the large sets $\mathcal{R}_l$, based on x (Lines 1 to 2). For each set pair $\langle R_i, R_j\rangle$ in $\mathcal{R}_l \times \mathcal{R}$, it adds the pair to the result set $\mathcal{A}$ if their intersection size is at least c (Lines 3 to 5). Next, for each small set $R_j \in \mathcal{R}_s$, it enumerates all its c-subsets,

	R_1	R_2	R_3	R_4	R_5	R_6	R_7
R_5	1	1	1	4	-	-	-
R_6	0	0	0	0	1	-	-
R_7	0	0	0	0	1	8	-

Figure 1: $\mathcal{R}_l \times \mathcal{R}$

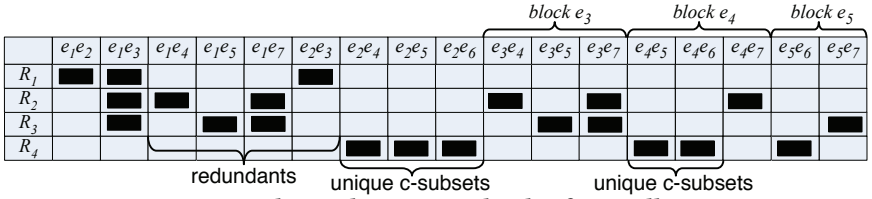


Figure 2: The c-subset inverted index for small sets.

and inserts them to the inverted index $\mathcal{L}$ (Lines 6 to 8). For each inverted list in $\mathcal{L}$, it adds every set pair in it to $\mathcal{A}$ (Lines 9 to 10). Finally, the algorithm returns $\mathcal{A}$ (Line 11).

Example 1. *Consider the dataset $\mathcal{R}$ in Table 1, and suppose that the threshold is $c = 2$. As explained in the next section, the size boundary is $x = 5$. Thus, we have $\mathcal{R}_s = \{R_1, R_2, R_3, R_4\}$ and $\mathcal{R}_l = \{R_5, R_6, R_7\}$. As shown in Figure 1, we enumerate every set pair in $\mathcal{R}_l \times \mathcal{R}$, and calculate their intersection size, which yields two similar pairs $\langle R_4, R_5\rangle$ and $\langle R_6, R_7\rangle$. Then, we build the inverted index for all the 2-subsets found in the small sets. The index is shown in Figure 2, where a black block indicates the existence of this c-subset in the corresponding small set. The inverted list $\mathcal{L}[\{e_1, e_3\}]$ has three sets R_1, R_2 and R_3, according to which we obtain three similar pairs $\langle R_1, R_2\rangle$, $\langle R_1, R_3\rangle$, and $\langle R_2, R_3\rangle$. Similarly, a similar pair is spawned from the inverted list $\mathcal{L}[\{e_1, e_7\}]$, and another from $\mathcal{L}[\{e_3, e_7\}]$. However, these two pairs have appeared earlier, and hence, are duplicates. In total, the join result consists of 5 pairs.*

Intuition Behind the Size Aware Algorithm. Existing approaches build an inverted index $\mathcal{I}$ for the elements in the sets. As will be discussed in Section 6.1, each inverted list $\mathcal{I}[e]$ is scanned $|\mathcal{I}[e]|$ times where $|\mathcal{I}[e]|$ is the inverted list length. Thus they need $O(|\mathcal{I}[e]|^2)$ time to process each inverted list $\mathcal{I}[e]$. Notice that, as there are n elements in total, the number of large sets cannot exceed $\frac{n}{x}$. Thus the large sets contribute at most $\frac{n}{x}$ to the inverted list length. On the contrary, there is no bound for the number of small sets and they could contribute up to $O(n)$ to the inverted list length which results in a time complexity of $O(n^2)$. This is why existing methods fail to perform effectively over small sets. This is also why we can afford using any existing methods to process large sets while have to design a new method for the small sets.

Remark. Obviously it is expensive to enumerate all c-subsets in every small set, especially when the small sets have large sizes. To address this issue, we propose various techniques to avoid enumerating a large number of them in Section 4 and our method has both theoretical and practical guarantees. In addition, any existing method can be plugged in our framework to process the large sets for better practical performance. In particular, we use `ScanCount` [30] as described in Section 6.1 in our implementation.

3.2 Size Boundary Selection in Theory

It remains to clarify the setting of the size boundary x (which divides the small sets from the large ones). We will adopt an analytic approach: bounding the running time of our algorithm as a function of x, and then finding the best x to minimize the cost.

Running Time as a Function of x: Let us first analyze the time complexity of finding the similar pairs in $\mathcal{R}_l \times \mathcal{R}$. Recall that our algorithm calculates the intersection size $|R \cap R'|$ for every pair of $\langle R, R'\rangle \in \mathcal{R}_l \times \mathcal{R}$. To do so efficiently, we create a hash table on every large set $R \in \mathcal{R}_l$ so that whether an element e belongs to R can be determined in constant time. Then, $|R \cap R'|$ can be computed in $O(|R'|)$ time, by probing the hash table of R with every element in R'. In other words, the computation of $|R \cap R'|$ for the same R' but all $R \in \mathcal{R}_l$ can be accomplished in $O(|R'| \cdot \frac{n}{x})$ time. Therefore, the size $|R \cap R'|$ of all $\langle R, R'\rangle \in \mathcal{R}_l \times \mathcal{R}$ can be obtained in time

$$\sum_{R' \in \mathcal{R}} O(|R'| \cdot \frac{n}{x}) \quad = \quad O\left(\frac{n^2}{x}\right).$$

We now proceed to discuss the time complexity of finding similar pairs in $\mathcal{R}_s \times \mathcal{R}_s$. There are two steps: (i) the first enumerates all the c-subsets to build the inverted index, and (ii) the second generates similar pairs from the inverted lists. A small set R has $\binom{|R|}{c}$ c-subsets. Since $|R| \leq x$ (as R is a small set), the total number of c-subsets from all small sets is at most:

$$\sum_{R \in \mathcal{R}_s} \binom{|R|}{c} \leq \sum_{R \in \mathcal{R}_s} |R|^c \leq x^{c-1} \sum_{R \in \mathcal{R}_s} |R| \leq x^{c-1} n.$$

The enumeration cost in the first step is asymptotically the same as the above number, i.e., the cost is bounded by $O(x^{c-1}n)$.

The cost of the second step comes from generating all the set pairs in each inverted list. Let $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_l$ be all the inverted lists in $\mathcal{L}$, and $|\mathcal{L}_i|$ be the length of $\mathcal{L}_i$. The time complexity of the second step is $O(\sum_{i=1}^{l} |\mathcal{L}_i|^2)$. As the total length of all the inverted lists is exactly the number of c-subsets in all the small sets, it holds that $\sum_{i=1}^{l} |\mathcal{L}_i| \leq x^{c-1} n$. Moreover, for any inverted list $\mathcal{L}_i$, we have $\frac{|\mathcal{L}_i|(|\mathcal{L}_i|-1)}{2} \leq k$ (remember that k is the total number of similar set pairs in $\mathcal{R} \times \mathcal{R}$) because the number of similar set pairs generated in $\mathcal{L}_i$ obviously cannot exceed k. It thus follows that $|\mathcal{L}_i| = O(\sqrt{k})$. Hence, the second step runs in time

$$O(\sum_{i=1}^{l} |\mathcal{L}_i|^2) = O(\sqrt{k} \sum_{i=1}^{l} |\mathcal{L}_i|) = O(x^{c-1} n \sqrt{k}).$$

Choosing x When k is Known: Let us first make an (unrealistic) assumption that we know in advance the value of k (the assumption will be removed shortly). In this scenario, the best value of x results directly from the earlier analysis. Specifically, as shown above, the overall running time of our algorithm is

$$O\left(\frac{n^2}{x} + x^{c-1} n \sqrt{k}\right).$$

To minimize the time complexity, we set $x = (n/\sqrt{k})^{1/c}$. In this case, the time complexity of our algorithm is

$$O\left(n^{2-\frac{1}{c}} k^{\frac{1}{2c}}\right). \tag{1}$$

As an example, consider the dataset in Table 1 again with the threshold $c = 2$. Here, $n = \sum_{i=1}^{7} |R_i| = 40$, and $k = 5$. Hence, we set $x = (40/\sqrt{5})^{\frac{1}{2}} \approx 5$.

When k is Not Known—The Doubling Trick: Now we return to the reality where one does not have the precise value of k. Interestingly, even in this case, it is still possible to achieve the same time

complexity as (1) with a technique often known as the *doubling trick*, which is a commonly used technique in theory for analyzing the complexity. The main idea is to guess k starting from a small value. Then, we run the algorithm *as if* our guess was accurate. If it is, then the algorithm indeed achieves the desired cost; otherwise, we are able to *detect* the fact that our guess is too low. In the former case, the join problem has already been solved, whereas in the latter, we double our guess for k and repeat. The algorithm eventually terminates; and when it does so, it is guaranteed that (i) our final guess is at most twice the real k, and that (ii) the total execution time is dominated by that of the last run (with the final guess).[1]

Next, we give the details of the above solution. The solution has multiple rounds. In each round we guess a k and execute the size aware algorithm (Algorithm 1). Let $\hat{k}$ be our guess of k in the current round. If the guess is accurate, we know from the earlier analysis that, the size aware algorithm must terminate by performing at most $\alpha \cdot n^{2-\frac{1}{c}} \hat{k}^{\frac{1}{2c}}$ "micro steps" (α is the hidden constant in the big-O of (1)), each of which takes $O(1)$ time, and can be tracked easily—more specifically, a micro step in our algorithm is one probe in a hash table in processing $\mathcal{R}_l \times \mathcal{R}$, or the enumeration of one set pair in processing $\mathcal{R}_s \times \mathcal{R}_s$. Therefore, as soon as $1 + \alpha \cdot n^{2-\frac{1}{c}} \hat{k}^{\frac{1}{2c}}$ micro steps have been performed, we know that our guess $\hat{k}$ is smaller than the real k. Hence, the size aware algorithm can now terminate itself—in which case, we say that the current round has finished. Then, we double $\hat{k}$ and perform another round until our guess $\hat{k} \geq k$. Note in each round it takes $O(n^{2-\frac{1}{c}} \hat{k}^{\frac{1}{2c}})$ time.

To achieve the desired complexity (1), we start with $\hat{k} = 1$. At its termination, the value of $\hat{k}$ is at most $2k$ (otherwise, the algorithm would have terminated in the previous round). Therefore, the overall running time of all the rounds is bounded by

$$O\left(\sum_{i=0}^{\log_2(2k)} n^{2-\frac{1}{c}} (2^i)^{\frac{1}{2c}}\right) = O\left(n^{2-\frac{1}{c}} k^{\frac{1}{2c}}\right).$$

We thus have proved:

Theorem 1. *There exists an algorithm with the time complexity $O(n^{2-\frac{1}{c}} k^{\frac{1}{2c}})$ for the set similarity join with overlap constraints problem, where n is the total size of all the sets, constant c is the similarity threshold, and k is the number of similar set pairs in the result.*

Beating the Quadratic Barrier: The value of k ranges from 0 to $\binom{n}{2}$. As explained in Section 1, we achieve sub-quadratic time whenever this is possible. That is, for $k = o(n^2)$, it always holds that $O(n^{2-\frac{1}{c}} k^{\frac{1}{2c}}) = o(n^2)$, whereas for $k = \Omega(n^2)$, any algorithm must spend $\Omega(n^2)$ time just to output all the similar pairs.

Remark: Note we make no assumptions about the distribution of the set sizes. In the extreme case where all the sets have exactly the same size ℓ, either all of them are classified as small sets, or all of them are classified as large sets, depending on the comparison between ℓ and $(n/\sqrt{k})^{1/c}$ (i.e., the size boundary). In both cases, the time complexity is as claimed—our proof holds in general.

Having proved the theoretical guarantee of our algorithm, in the subsequent sections, we will strive to improve its *practical* performance dramatically with careful optimization heuristics. Focus will be placed on processing $\mathcal{R}_s \times \mathcal{R}_s$, as any existing approach can be used to process $\mathcal{R}_l \times \mathcal{R}$. As a serious challenge, our current algorithm needs to enumerate all the c-subsets of a small set, the number of which can be huge, thus causing significant overhead. We will remedy this issue with novel ideas, as presented below.

[1]Note the doubling trick is only for the complexity analysis. In practice, we use the approach later proposed in Section 5 to determine the size boundary.

4 HEAP-BASED METHODS ON SMALL SETS

In this section, we focus on building the inverted index $\mathcal{L}_{slim}$ for c-subsets in $\mathcal{R}_s$ that can generate all the results in $\mathcal{R}_s \times \mathcal{R}_s$, which we shall call a *slimmed inverted index*, instead of the full inverted index $\mathcal{L}$. It is possible to skip some unnecessary c-subsets in $\mathcal{R}_s$ when we construct a slimmed inverted index, which includes *unique* c-subsets and *redundant* c-subsets. We propose heap-based methods to skip unique and redundant c-subsets in Section 4.1 and Section 4.2 respectively. As it is expensive to maintain the heap, especially when the heap is wide, we propose a blocking-based method to shrink the heap in Section 4.3.

4.1 Skipping Unique c-subsets

For each small set, the size-aware algorithm needs to enumerate all its c-subsets to build the full inverted index $\mathcal{L}$ and generate the results based on it. If a c-subset is unique, i.e., it appears only once in all the small sets, we can avoid generating it and get a slimmed inverted index that can generate all the results as the unique c-subset cannot produce any result.

Definition 2 (Unique c-subset). *A c-subset $\mathtt{r}_c$ is called a unique c-subset if $|\mathcal{L}[\mathtt{r}_c]| = 1$.*

As there are a large number of unique c-subsets, it is important to avoid generating them. For example, in Figure 2, R_4 has 6 c-subsets and all of them are unique c-subsets. Thus we do not need to generate them. Given a set R, it has $\binom{|R|}{c}$ c-subsets and it is prohibitively expensive to generate all of them. Fortunately, most of them are unique c-subsets and next we discuss how to skip them.

Skip Unique c-subsets. We first give the basic idea of skipping unique c-subsets. We fix a global order for the c-subsets in all the small sets and visit the c-subsets in ascending order. As shown in Figure 3, consider a c-subset $\mathtt{r}_c$ in a small set R. Let $\mathtt{r}'_c$ be the smallest c-subset that is larger than $\mathtt{r}_c$ in $\mathcal{R}_s \setminus \{R\}$ (i.e., not in R). Then all the c-subsets between $\mathtt{r}_c$ and $\mathtt{r}'_c$ (the gray ones in the figure) must only appear in R and must be unique c-subsets (this is based on the definition of $\mathtt{r}'_c$). Thus we can skip the c-subsets in R which are larger than $\mathtt{r}_c$ and smaller than $\mathtt{r}'_c$. Next we discuss how to utilize this idea to skip unique c-subsets.

Global Ordering. We fix a global order for all the *elements* in the small sets and sort the elements in each small set by this global order. Then we can order the c-subset based on the order of elements, i.e., first by the smallest element, then by the second smallest element and finally by the largest element. For example, consider the four small sets in Table 1 and suppose that we order the elements by their subscripts, i.e., the order of $e_1, e_2, \ldots, e_7$. Then the order of the 2-subsets is shown in Figure 2 from left to right.

Heap-based Method. We first give a naive heap-based method to construct the entire inverted index $\mathcal{L}$. For each small set, we visit its c-subsets in ascending order and denote the smallest unvisited c-subset as its *min-subset*. A min-heap $\mathcal{H}$ is used to manage all the

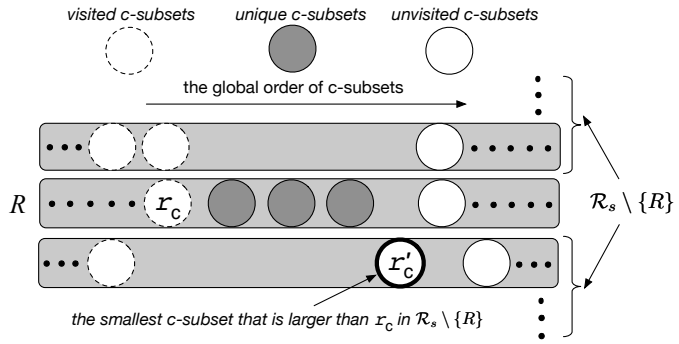


Figure 3: Skip the unique c-subsets.

min-subsets of the small sets. We pop $\mathcal{H}$ to get the globally smallest min-subset, which is denoted as r_c^{min}. Suppose that r_c^{min} comes from the set R. We append R to the inverted list $\mathcal{L}[\mathsf{r}_c^{min}]$, mark r_c^{min} as visited and reinsert the next min-subset of R to the heap. Iteratively, we can build all the inverted lists and get the entire inverted index $\mathcal{L}$.

Next we construct a slimmed inverted index by excluding the inverted lists of the unique c-subsets from $\mathcal{L}$. For this purpose, every time we pop the heap and get the smallest min-subset r_c^{min} from a small set R, we can again compare r_c^{min} with the min-subset that currently tops the heap, which is denoted as r_c^{top}. If $\mathsf{r}_c^{top} \neq \mathsf{r}_c^{min}$, instead of reinserting the next min-subset of R to the heap, we can jump directly to the smallest c-subset in R that is no smaller than r_c^{top} and reinsert it to the heap as the skipped c-subsets must only appear in R and must be unique c-subsets (recall the basic idea above, in which case r_c^{min} corresponds to r_c and r_c^{top} corresponds to r'_c). We can achieve this by a binary search as the elements and c-subsets in R are ordered. The details of the binary search are described in Appendix A.

The pseudo code of the HeapSkip method is shown in Algorithm 2. Instead of enumerating every c-subset in each small set, it first fixes a global order for all the elements and builds a min-heap $\mathcal{H}$ by inserting all the min-subsets of the small sets to $\mathcal{H}$ (Lines 1 to 2). It keeps popping $\mathcal{H}$ until it is empty (Lines 3 to 9). Suppose that the smallest popped out min-subset r_c^{min} comes from R, it appends R to the inverted list $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$ and compares r_c^{min} with r_c^{top} which is the current top element of $\mathcal{H}$. If r_c^{min} and r_c^{top} are different, it binary searches the first c-subset in R that is no smaller than r_c^{top} and reinserts it to $\mathcal{H}$ (Lines 6 to 7); otherwise it reinserts the next min-subset in R into $\mathcal{H}$ (Line 9). Finally, it returns a slimmed inverted index $\mathcal{L}_{slim}$ (Line 10).

Example 2. *Consider the dataset $\mathcal{R}$ in Table 1 and suppose the threshold is $c = 2$. There are 4 small sets, R_1, R_2, R_3, and R_4. As illustrate in Figure 2,* HeapSkip *first orders the elements in them by their subscripts. Then it inserts the min-subsets $\{e_1, e_2\}, \{e_1, e_3\}, \{e_1, e_3\}$, and $\{e_2, e_4\}$ of the four small sets into a min-heap $\mathcal{H}$. Next it pops $\mathcal{H}$ and has $\mathsf{r}_c^{min} = \{e_1, e_2\}$ from R_1 and $\mathsf{r}_c^{top} = \{e_1, e_3\}$. It appends R_1 to the inverted list $\mathcal{L}_{slim}[\{e_1, e_2\}]$. As $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$, it binary searches the first c-subset in R_1 that is no smaller than r_c^{top}. It gets $\{e_1, e_3\}$ and reinserts this c-subset to $\mathcal{H}$. Then it pops $\mathcal{H}$ and has $\mathsf{r}_c^{min} = \{e_1, e_3\}$ from R_1 and $\mathsf{r}_c^{top} = \{e_1, e_3\}$. It appends R_1 to $\mathcal{L}_{slim}[\{e_1, e_3\}]$. As $\mathsf{r}_c^{min} = \mathsf{r}_c^{top}$, it reinserts the next min-subset $\{e_2, e_3\}$ of R_1 to $\mathcal{H}$. Iteratively, it can build a slimmed inverted index $\mathcal{L}_{slim}$. Note it can skip the unique c-subsets $\{e_2, e_5\}$ and $\{e_2, e_6\}$ of R_4 by binary searching the smallest c-subset in R_4 that is no smaller than $\mathsf{r}_c^{top} = \{e_3, e_4\}$ when $\mathsf{r}_c^{min} = \{e_2, e_4\}$ comes from R_4. Similarly it can also skip the c-subset $\{e_4, e_6\}$ of R_4 when $\mathsf{r}_c^{min} = \{e_4, e_5\}$ and $\mathsf{r}_c^{top} = \{e_4, e_7\}$.*

Algorithm 2: HeapSkip

Input: $\mathcal{R}_s$: all the small sets; c: threshold;
Output: $\mathcal{L}_{slim}$: a slimmed inverted index for $\mathcal{R}_s$;

```
1  Fix a global order for all the elements in R_s;
2  Insert all the min-subsets of small sets to a heap H;
3  while H is not empty do
4      pop H to get r_c^min and suppose it is from R;
5      append R to L_slim[r_c^min];
6      if r_c^top ≠ r_c^min then
7          binary search for the first c-subset in R that is no
           smaller than r_c^top and reinsert it into H;
8      else
9          reinsert the next min-subset in R into H;
10 return L_slim
```

4.2 Skipping Redundant c-subsets

For small sets, the size-aware algorithm may produce duplicate results as some set pairs may share multiple common c-subsets. If a c-subset only generates duplicate results, we can skip enumerating it and still get a slimmed inverted index. Obviously, given two c-subsets r_c and r'_c, if $\mathcal{L}[\mathsf{r}_c] \subseteq \mathcal{L}[\mathsf{r}'_c]$, then r_c is redundant, because the result generated by r_c (i.e., $\mathcal{L}[\mathsf{r}_c] \times \mathcal{L}[\mathsf{r}_c]$) is a subset of that generated by r'_c (i.e., $\mathcal{L}[\mathsf{r}'_c] \times \mathcal{L}[\mathsf{r}'_c]$).

Definition 3 (Redundant c-subset). *A c-subset r_c is a redundant c-subset of another c-subset r'_c if $\mathcal{L}[\mathsf{r}_c] \subseteq \mathcal{L}[\mathsf{r}'_c]$.*

Note that the duplicate results are generated whenever $|\mathcal{L}[\mathsf{r}_c] \cap \mathcal{L}[\mathsf{r}'_c]| \geq 2$. However, it is expensive to eliminate all the duplicate results. In fact, it remains expensive to detect all redundant c-subsets, as it requires to enumerate every two c-subsets and checks whether the inverted list of one c-subset is a subset of the other. To address this issue, we propose an efficient algorithm that can detect all *adjacent redundant c-subsets*.

Definition 4 (Adjacent Redundant c-subset). *The c-subset r_c is an adjacent redundant c-subset of another c-subset r'_c if $\mathsf{r}'_c \prec \mathsf{r}_c$, where $\prec$ denotes the order of c-subsets, and the c-subsets between r'_c and r_c, including r_c, are all redundant c-subsets of r'_c.*

For example, in Figure 2, we have $\mathcal{L}[\{e_1e_3\}] = \{R_1, R_2, R_3\}$, $\mathcal{L}[\{e_1e_4\}] = \{R_2\}$, $\mathcal{L}[\{e_1e_5\}] = \{R_3\}$, $\mathcal{L}[\{e_1e_7\}] = \{R_2, R_3\}$, and $\mathcal{L}[\{e_2e_3\}] = \{R_1\}$. Thus, based on the definition, $\{e_1e_4\}$, $\{e_1e_5\}$, $\{e_1e_7\}$, and $\{e_2e_3\}$ are all adjacent redundant c-subsets of $\{e_1e_3\}$. If we skip them, we can build a smaller slimmed inverted index.

Skip Adjacent Redundant c-subsets. We first give the basic idea of skipping adjacent redundant c-subsets. As shown in Figure 4, we still visit the c-subsets in ascending order. Let r''_c be the smallest c-subset that is larger than r_c in $\mathcal{R}_s \setminus \mathcal{L}[\mathsf{r}_c]$ (i.e., sets not containing r_c). We find that all the c-subsets in the small sets in $\mathcal{L}[\mathsf{r}_c]$ that are larger than r_c and smaller than r''_c (the gray ones in the figure, e.g., r'_c) are adjacent redundant c-subsets of r_c as their inverted lists are all sub-lists of $\mathcal{L}[\mathsf{r}_c]$. Next we discuss how to utilize this idea to skip adjacent redundant c-subsets.

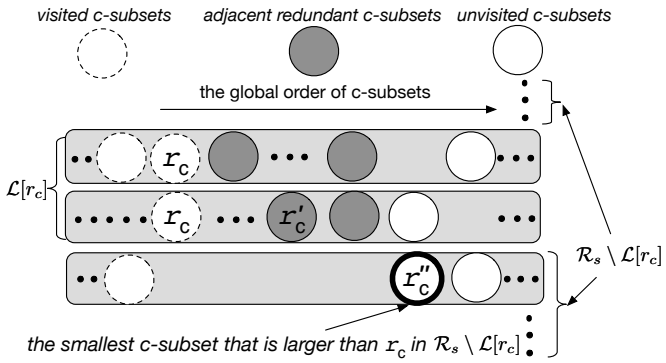


Figure 4: Skip the adjacent redundant c-subsets.

Heap-based Method. We still fix the order of the c-subsets by the order of elements, access the c-subsets of each set in order, utilize a min-heap $\mathcal{H}$ to manage the min-subsets, and iteratively pop the min-heap to build the inverted lists. Every time we pop $\mathcal{H}$ and get r_c^{min} from a set R, we first append R to $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$. Then we compare r_c^{min} with r_c^{top}. However, if $\mathsf{r}_c^{min} = \mathsf{r}_c^{top}$, we do not reinsert the next min-subset of R to $\mathcal{H}$. Only if $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ we reinsert the min-subsets of all the sets in $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$ to $\mathcal{H}$ by binary searching the first c-subsets that are no smaller than r_c^{top}. In this way, we can skip those c-subsets larger than r_c^{min} and smaller than r_c^{top} which must be adjacent redundant c-subsets of r_c^{min} as the inverted lists of these c-subsets are all sub-lists of $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$ (recall the basic idea above, in which case r_c^{min} corresponds to r_c and r_c^{top} corresponds to r_c'').

The pseudo code of the HeapDedup method is shown in Algorithm 3. HeapDedup improves on HeapSkip by lazily reinserting the min-subsets. Instead of reinserting a min-subset to the min-heap every time, HeapDedup reinserts a batch of min-subsets to the min-heap by binary searching the min-subsets no smaller than r_c^{top} when $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ (Lines 1 to 3) and does nothing when $\mathsf{r}_c^{min} = \mathsf{r}_c^{top}$.

Example 3. *Consider the four small sets R_1, R_2, R_3 and R_4 in Table 1 and suppose that the threshold is $c = 2$. As illustrate in Figure 2,* HeapDedup *first inserts the min-subsets $\{e_1, e_2\}$, $\{e_1, e_3\}$, $\{e_1, e_3\}$ and $\{e_2, e_4\}$ of the four small sets into a min-heap $\mathcal{H}$. Next it pops $\mathcal{H}$, gets $\mathsf{r}_c^{min} = \{e_1, e_2\}$ from R_1 and reinserts the next r_c^{min} $\{e_1, e_3\}$ of R_1 to $\mathcal{H}$. Then it pops $\mathcal{H}$ again and has $\mathsf{r}_c^{min} = \{e_1, e_3\}$ from R_1 and $\mathsf{r}_c^{top} = \{e_1, e_3\}$. It appends R_1 to $\mathcal{L}_{slim}[\{e_1, e_3\}]$. As $\mathsf{r}_c^{min} = \mathsf{r}_c^{top}$, it keeps popping $\mathcal{H}$ and has $\mathsf{r}_c^{min} = \{e_1, e_3\}$ from R_2 and $\mathsf{r}_c^{top} = \{e_1, e_3\}$. It appends R_2 to $\mathcal{L}_{slim}[\{e_1, e_3\}]$. As $\mathsf{r}_c^{min} = \mathsf{r}_c^{top}$, it pops $\mathcal{H}$ and has $\mathsf{r}_c^{min} = \{e_1, e_3\}$ from R_3 and $\mathsf{r}_c^{top} = \{e_2, e_4\}$. It appends R_3 to $\mathcal{L}_{slim}[\{e_1, e_3\}]$. As $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$, for the sets R_1, R_2, and R_3 in $\mathcal{L}_{slim}[\{e_1, e_3\}]$, it binary searches the first min-subsets in them that are no smaller than r_c^{top}. It gets $\{e_3, e_4\}$ and $\{e_3, e_5\}$ for R_2 and R_3 and reinserts them to $\mathcal{H}$. It reaches the end of R_1 and does not reinsert any c-subset for R_1 to $\mathcal{H}$. Iteratively it can build a slimmed inverted index without adjacent redundant c-subsets.*

4.3 Blocking c-subsets

For each small set, the heap-based methods need to maintain a min-subset in the min-heap. Thus the heap size is $|\mathcal{R}_s|$, which is rather large and leads a high heap adjusting cost (the time cost for each heap adjusting operation is $c \times \log |\mathcal{R}_s|$ as each c-subset comparison takes c cost).

Algorithm 3: HeapDedup

Input: $\mathcal{R}_s$: all the small sets; c: threshold;
Output: $\mathcal{L}_{slim}$: a slimmed inverted index for $\mathcal{R}_s$;
// replace lines 6 to 9 of Algorithm 2
1 **if** $\mathsf{r}_c^{top} \neq \mathsf{r}_c^{min}$ **then**
2 **foreach** R *in* $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$ **do**
3 binary search the first c-subset in R that is no smaller than r_c^{top} and reinsert it to $\mathcal{H}$;
4 **else**
5 **continue**; // lazy reinsertion

To address this issue, we propose to block the c-subsets by their smallest elements. As shown in Figure 5, consider the block $\mathcal{B}_e$ with smallest element e. As the other c-subsets either have the smallest elements larger than e or smaller than e, they must be different from the c-subsets in the block $\mathcal{B}_e$. Thus we can independently utilize the heap-based methods to build a part of the slimmed inverted index for the c-subsets in $\mathcal{B}_e$ with a smaller heap (as we do not need to maintain the min-subsets for those small sets without c-subsets in $\mathcal{B}_e$, such as R_a and R_b in the figure). Next we formalize our idea.

We first fix a global order for all the elements. Then we build an inverted index $\mathcal{I}$ for all the elements in $\mathcal{R}_s$ to facilitate blocking the c-subsets. The inverted list $\mathcal{I}[e]$ of the element e consists of all the small sets containing e. As all the c-subsets in a block $\mathcal{B}_e$ must contain the element e while all the small sets having element e are in $\mathcal{I}[e]$, the c-subsets in the block $\mathcal{B}_e$ are from and only from the sets in $\mathcal{I}[e]$. Thus for each inverted list $\mathcal{I}[e]$, we apply the heap-based method on all the sets in $\mathcal{I}[e]$ to construct the inverted list $\mathcal{L}_{slim}[\mathsf{r}_c]$ for every c-subset $\mathsf{r}_c \in \mathcal{B}_e$. Note we only need to access those c-subsets with the smallest element e in the sets in $\mathcal{I}[e]$. To achieve this, we can perform a simulation by removing those elements no larger than e in the sets in $\mathcal{I}[e]$ and decreasing the threshold by 1 when applying the heap-based methods.[2]

The pseudo code of the BlockDedup is shown in Algorithm 4. It first fixes a global order for elements and then builds the element inverted index $\mathcal{I}$ (Lines 1 to 2). Next for each inverted list $\mathcal{I}[e] \in \mathcal{I}$, it generates a temporary set R_{tmp} of sets by removing all the elements no larger than e in the sets in $\mathcal{I}[e]$ (Line 4)[3]. Then it applies the HeapDedup method on R_{tmp} with the threshold $c - 1$ to build a part of the slimmed inverted index $\mathcal{L}_{slim}$ (Line 5). Note the blocking-based method can also work with HeapSkip here and is named as BlockSkip in the experiment. Finally it returns a slimmed inverted index $\mathcal{L}_{slim}$ (Line 6).

Example 4. *Consider the small sets in Table 1 and suppose that the threshold $c = 2$. In Figure 2, we can group all their c-subsets to 5 blocks. The c-subsets with e_1, e_2, e_3, e_4, and e_5 as their smallest element respectively. The block of e_3 contains three c-subsets, $\{e_3, e_4\}$, $\{e_3, e_5, \}$, and $\{e_3, e_7\}$. Note the c-subset $\{e_1, e_3\}$ is not belong to this block as its minimum element is e_1 rather than e_3. The block of e_3 only has c-subsets from two small sets, R_2 and R_3. We can utilize the*

[2] In our implementation, we do not remove the elements and copy all the sets. Instead we omit the elements no larger than e when accessing the c-subsets in heap-based methods.
[3] If the size of a set is smaller than $c - 1$ after removing the elements no larger than e, we do not need to add it into R_{tmp}. Instead, we drop it.

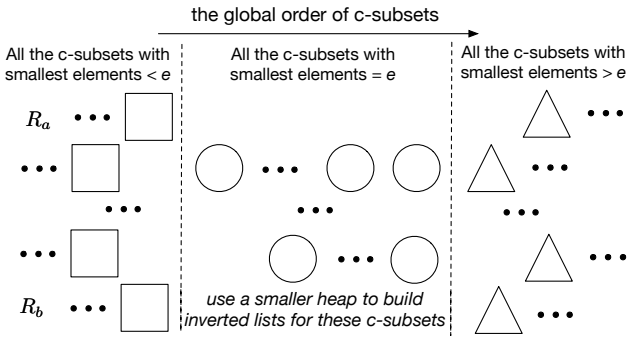


Figure 5: Block c-subsets with smallest element e.

heap-based methods on this block with a smaller heap size of 2 *to build a part of a slimmed inverted index.*

5 SIZE BOUNDARY SELECTION IN PRACTICE

The complexity analysis in Section 3 gives us the insight that we need to process the small and large sets separately. It gives the size boundary by equating the time complexities of the small and large sets. However there is a gap between the time complexity and the actual time cost. In practice, the number of the enumerated c-subsets is much smaller than $x^{c-1}n$ due to the pruning techniques in Section 4 and the upper bounds used in analyzing the total number of c-subsets under the worst case in Section 3. Moreover, the lengths of the inverted lists of the c-subsets are much shorter than $\sqrt{k}$ in practice and the time cost for generating the results is far smaller than $x^{c-1}n\sqrt{k}$. Thus the time complexity largely overestimates the time cost for processing the small sets and the suggested size boundary $(\frac{n}{\sqrt{k}})^{\frac{1}{c}}$ is too small in practice.

Next we give the basic idea of our size boundary selection method. Based on the time complexity analysis, with the increasing of the size boundary x, the time complexity of the small sets $O(x^{c-1}n\sqrt{k})$ grows more and more sharply while the time complexity of the large sets $O(\frac{n^2}{x})$ falls less and less precipitously. Thus we can increase the size boundary from the smallest set size in $\mathcal{R}$ and estimate the time costs for the small sets and the large sets. We stop increasing the boundary when the time cost for small sets grows more than the decrease of the time cost for large sets, and partition all the sets by the current size boundary. To this end, we show how to estimate the time costs for the small sets and the large sets in Section 5.1 and propose an effective size boundary selection method in Section 5.2.

5.1 Estimating the Time Costs

Next we estimate the costs for processing large sets and small sets.

Estimating the time cost for large sets: In our implementation, we use the `ScanCount` [13] method to process the large sets. For the large sets, we build an inverted index $\mathcal{I}$ for all the elements in the sets in $\mathcal{R}$. For each large set $R \in \mathcal{R}_l$, we scan the corresponding inverted lists of its elements and count the occurrences of the other sets in the inverted lists. All the sets with occurrence times no smaller than c are similar to R. We can estimate the time cost for processing the large set R by adding up the lengths of all its inverted lists. Thus the time cost for all the large sets is proportional to $\sum_{R\in\mathcal{R}_l}\sum_{e\in R}|\mathcal{I}[e]|$. We can get this cost by scanning the entire dataset for one pass.

Estimating the time cost for small sets. For the small sets, the size-aware method uses a heap to manage the min-subsets, accesses the c-subsets of each small set by binary searching, and generates the results by scanning the c-subset inverted index. Thus there are three major costs, the heap adjusting cost, the binary searching cost, and the result generation cost. Next we estimate them.

We first estimate the result generation cost. Obviously the result generation cost is proportional to the number of c-subsets shared by the small sets as each set pair generated from the c-subset inverted index corresponds to a c-subset shared by the set pair and any small set pair sharing a c-subset corresponds to two entries in the inverted list of this c-subset. Thus we can randomly sample y small set pairs from all the $Y = \binom{|\mathcal{R}_s|}{2}$ small set pairs. Suppose that for the i^{th} sampling set pair, they share p_i common elements; then they share $\binom{p_i}{c}$ c-subsets. Based on the law of the large numbers, we can estimate the result generation cost as proportional to $\frac{Y}{y}\sum_{i=1}^{y}\binom{p_i}{c}$ as the total number of small set pairs Y is large.

Next we estimate the heap adjusting cost and the binary search cost. The size-aware method blocks the c-subsets based on their smallest elements and utilizes the heap-based methods to process each block. There are a large number of distinct elements, and the number of blocks is also large (the number of blocks is the same as the number of distinct elements). We can randomly sample a number of blocks to estimate the heap adjusting cost and the binary searching cost for all the blocks. More specifically, for each sample block, we run the heap-based method. For each heap adjusting operation, we estimate its cost as proportional to $c \log h$ where h is the current heap size. For each binary search operation, we estimate its cost as proportional to $c \log t$ where t is the size of the set on which we do a binary search. Suppose that we randomly sample z blocks out of all $Z = |\mathcal{I}|$ blocks and have the heap adjusting cost for the i^{th} block is proportional to H_i and the binary search cost is proportional to T_i; then based on the law of large numbers, we can estimate that the heap adjusting cost and binary searching cost for all the blocks are proportional to $\frac{Z}{z}\sum_{i=1}^{z}(H_i + T_i)$ as the number of blocks Z is quite large.

Algorithm 4: BlockDedup

Input: $\mathcal{R}_s$: all the small sets; c: threshold;
Output: $\mathcal{L}_{slim}$: a slimmed inverted index for $\mathcal{R}_s$;

```
Fix a global order for all the elements in R_s;
Build an inverted index I for all the elements in R_s;
foreach I[e] in I do
    R_tmp = sets in I[e] with elements ≤ e removed;
    L_slim = L_slim ∪ HeapDedup(R_tmp, c − 1);
    // L_slim = L_slim ∪ HeapSkip(R_tmp, c − 1) for
       BlockSkip
return L_slim
```

5.2 The Size Boundary Selection Method

In this section, we propose a size boundary selection method. Based on Section 3.2, the time complexities of the small sets and the large sets are respectively $O(x^{c-1}n\sqrt{k})$ and $O(\frac{n^2}{x})$. The slope of $x^{c-1}n\sqrt{k}$ is always positive and monotonically increasing[4] w.r.t. the size boundary x while the slope of $\frac{n^2}{x}$ is always negative and also

[4]When $c = 2$, though the slope is a constant, the size boundary selection method proposed presently still works.

Algorithm 5: GetSizeBoundary

Input: $\mathcal{R}$: the dataset; c: the threshold;
Output: x: a size boundary for dichotomizing $\mathcal{R}$;

1 Set x as the larger of the smallest set size in $\mathcal{R}$ and c;
2 Estimate the time cost for $\mathcal{R}_s^x$ as $\mathcal{Z}'$ and for $\mathcal{R}_l^x$ as $\mathcal{Y}'$;
3 **while** *x is no larger than the largest set size in $\mathcal{R}$* **do**
4 Estimate the time cost $\mathcal{Z}$ for the small sets $\mathcal{R}_s^{x+1}$;
5 Estimate the time cost $\mathcal{Y}$ for the large sets $\mathcal{R}_l^{x+1}$;
6 **if** *benefit* $= \mathcal{Y}' - \mathcal{Y} \leq$ *cost* $= \mathcal{Z} - \mathcal{Z}'$ **then break**
  $x = x + 1$, $\mathcal{Y}' = \mathcal{Y}$ and $\mathcal{Z}' = \mathcal{Z}$;
7 **return** x

monotonically increasing w.r.t. the size boundary x. This means with the increasing of the size boundary x, the time complexity of the small sets grows first slowly and then sharply while the time complexity of the large sets falls first precipitously and then slowly. Based on this idea, we propose a cost model which uses the decrease of the time cost for the large sets as the benefit and the increase of the time cost for the small sets as the cost. More specifically, we first set the size boundary x as the smallest set size in $\mathcal{R}$ or the threshold c, whichever is larger and try to increase x by 1 each time. Let $\mathcal{R}_s^x$ and $\mathcal{R}_l^x$ respectively be the sets of small sets and large sets achieved by the size boundary x. Then we estimate the time costs for $\mathcal{R}_s^x$, $\mathcal{R}_s^{x+1}$, $\mathcal{R}_l^x$, and $\mathcal{R}_l^{x+1}$ as proportional to $\mathcal{Z}'$, $\mathcal{Z}$, $\mathcal{Y}'$, and $\mathcal{Y}$. Next we compare the benefit, which is proportional to $\mathcal{Y}' - \mathcal{Y}$, with the cost, which is proportional to $\mathcal{Z} - \mathcal{Z}'$. If the benefit is larger than the cost, we increase the size boundary x by 1 and repeat this procedure. Otherwise, we stop increasing x and dichotomize $\mathcal{R}$ by this size boundary. Note this method makes no assumptions about the distribution of the set sizes. If all the sets have the same size, it will classify all the sets either as small or large, depending on the estimations.

The pseudo-code of the cost-based method is shown in Algorithm 5. It takes a dataset $\mathcal{R}$ and a threshold c as input and outputs a size boundary x for dichotomizing $\mathcal{R}$. It first sets x to the larger one of the smallest set size in $\mathcal{R}$ and c (Line 1). Then it estimates the time costs $\mathcal{Z}'$ and $\mathcal{Y}'$ for $\mathcal{R}_s^x$ and $\mathcal{R}_l^x$ (Line 2), and the time costs $\mathcal{Z}$ and $\mathcal{Y}$ for $\mathcal{R}_s^{x+1}$ and $\mathcal{R}_l^{x+1}$ (Lines 4 to 5). If the benefit $\mathcal{Y}' - \mathcal{Y}$ is smaller than the cost $\mathcal{Z} - \mathcal{Z}'$, it stops and returns x as the size boundary (Line 6). Otherwise it increases x by 1, sets $\mathcal{Y}'$ to $\mathcal{Y}$ and $\mathcal{Z}'$ to $\mathcal{Z}$ (Line 6) and repeats the estimation until x is larger than the largest set size in $\mathcal{R}$ (Line 3).

Example 5. *Consider the dataset $\mathcal{R}$ in Table 1 and suppose that the threshold is $c = 3$. The complexity analysis suggests the size boundary as $(\frac{40}{\sqrt{3}})^{\frac{1}{3}} = 2.8$ and then all the sets are large sets. Our method first sets the boundary size x as the smallest set size 3. $\mathcal{R}_s^3$ is empty and $\mathcal{R}_l^3$ has all the sets. The time cost $\mathcal{Z}'$ for $\mathcal{R}_s^3$ is 0 while the cost $\mathcal{Y}'$ for $\mathcal{R}_l^3$ is 67. $\mathcal{R}_s^4$ contains R_1 and $\mathcal{R}_l^4$ has the other sets. The time cost $\mathcal{Z}$ for $\mathcal{R}_s^4$ is still 0 while the cost $\mathcal{Y}$ for $\mathcal{R}_l^4$ is 64. As the benefit $\mathcal{Y}' - \mathcal{Y} = 3$ is larger than the cost $\mathcal{Z} - \mathcal{Z}' = 0$, we increase x to 4 and set $\mathcal{Y}' = 64$ and $\mathcal{Z}' = 0$. Then $\mathcal{R}_s^5 = \{R_1, R_2, R_3, R_4\}$ and $\mathcal{R}_l^5 = \{R_5, R_6, R_7\}$. The cost $\mathcal{Y}$ for $\mathcal{R}_l^5$ is 42 while the cost $\mathcal{Z}$ for $\mathcal{R}_s^5$ is 55. As the benefit $\mathcal{Y} - \mathcal{Y}' = 22$ is smaller than the cost $\mathcal{Z}' - \mathcal{Z} = 55$, we stop increasing x and set $x = 4$.*

Table 2: The dataset details

	$\lvert\mathcal{R}\rvert$	n	avg, min, max $\lvert R\rvert$			$\lvert\mathcal{I}\rvert$	avg, min, max $\lvert\mathcal{I}[e]\rvert$		
DBLP	1M	10M	10.1	1	304	183K	55228	1	183226
CLICK	0.99M	8M	8.1	1	2,498	41K	194	1	601374
ORKUT	1M	77M	77.1	1	27,317	2.9M	2822	1	10785
ADDRESS	1M	7M	7	7	7	657K	10.65	1	223321

6 EXPERIMENTS

This section evaluates the efficiency and scalability of our methods.

6.1 Setup

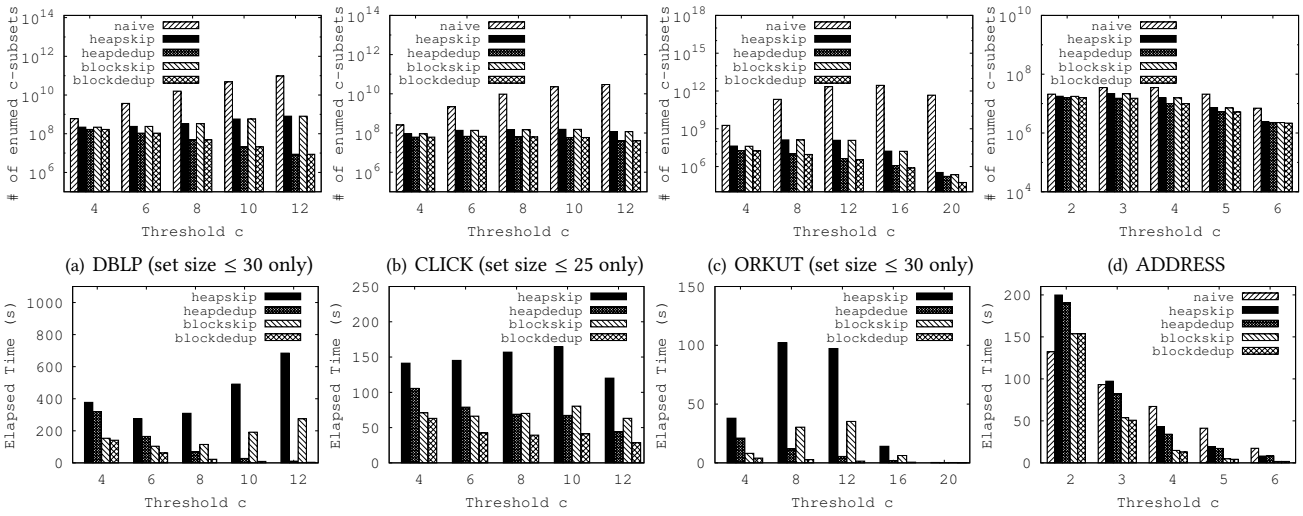
We implemented all our proposed techniques and conducted experiments on four real-world datasets DBLP[5], CLICK[6], ORKUT[7], and ADDRESS[8]. DBLP is a bibliography dataset from DBLP where each title is a set and each word is an element. CLICK is an anonymized click-stream dataset from a Hungarian on-line news portal where each set is a user and each click record is an element. ORKUT is a social network dataset from Orkut, where each set corresponds to a user and each element is a friend of the user. The friendship relation is undirected such that if two users are friends, they appear in each other's set. We randomly selected 1 million sets from DBLP and ORKUT and almost 1 million sets from CLICK as our datasets. ADDRESS is a collection of addresses crawled from the CSV tables on www.data.gov, where each element is a whitespace-delimited word. We randomly chose 1 million sets with exactly 7 elements to verify that our method worked well for the sets with the same sizes. In the experiment, our size boundary selection method classified all the sets in ADDRESS as small sets. We did not use a dataset in which the sets are of the same size and are all classified as large since `SizeAware` will use an existing method to process them. In this case `SizeAware` is the same as the existing method and the results are less interesting. The detailed information of all the datasets are shown in Table 2. We show their size distributions in Appendix E.

We compared our size-aware algorithm with the following state-of-the-art approaches for set similarity join with overlap constraints.

`ScanCount` [13]: It first builds an inverted index $\mathcal{I}$ for all the sets in a given dataset $\mathcal{R}$, where each entry is an element in the sets and is associated with an inverted list, which keeps all the sets that contain the element. Let $\mathcal{I}[e]$ denote the inverted list of the element e. $\mathcal{I}[e]$ consists of all the sets containing e. For example, for the dataset $\mathcal{R}$ in Table 1, we have $\mathcal{I}[e_2] = (R_1, R_4, R_5)$. For each set R, the `ScanCount` method scans all the corresponding inverted lists of its elements and counts the occurrence of each set in these inverted lists. Then it outputs all the sets with occurrences no smaller than c as similar sets of R. Let $|\mathcal{I}[e]|$ denote the length of the inverted list $\mathcal{I}[e]$. For each set in $\mathcal{I}[e]$, the set contains element e and this method needs to scan $\mathcal{I}[e]$. As $\mathcal{I}[e]$ has $|\mathcal{I}[e]|$ elements, $\mathcal{I}[e]$ is scanned $|\mathcal{I}[e]|$ times. Thus the time complexity is $O(\sum_{\mathcal{I}[e]\in\mathcal{I}} |\mathcal{I}[e]|^2) = O(n^2)$. Note that our `SizeAware` uses this method to process large sets.

`DivideSkip` [13]: Same as the `ScanCount` method, `DivideSkip` also builds an inverted index for the elements in the sets. However, for each set R, instead of scanning all the corresponding inverted lists of its elements, `DivideSkip` first scans some relatively shorter inverted lists to generate candidates and then binary searches the other longer inverted lists to get the finally results[9]. This method still has the worst-case time complexity of $O(n^2)$.

[5] http://dblp.uni-trier.de/
[6] http://fimi.cs.helsinki.fi/data/
[7] https://snap.stanford.edu/data/com-Orkut.html
[8] http://www.data.gov
[9] It divides the short and long inverted lists based on a heuristic [13].



(a) DBLP (set size ≤ 30 only) (b) CLICK (set size ≤ 25 only) (c) ORKUT (set size ≤ 30 only) (d) ADDRESS

(e) DBLP (set size ≤ 30 only) (f) CLICK (set size ≤ 25 only) (g) ORKUT (set size ≤ 30 only) (h) ADDRESS

Figure 6: Evaluating the Heap-based Methods

`All-Pairs` [3]: `All-Pairs` first fixes a global order for all the distinct elements in $\mathcal{R}$, such as the alphabetical order or the frequency order. Then it sorts the elements in each set by this global order and generates the prefix of each set, where the prefix of the set R consists of its first $|R| - c + 1$ elements. It can guarantee that two sets are similar only if their prefixes share at least one common element. Next `All-Pairs` builds an inverted index for all the elements in the prefixes. For each set R, it unions all the inverted lists of the elements in the prefix of R as candidates and verifies them by calculating their real similarity to R. Its time complexity is $O(n^2)$.[10]

`AdaptJoin` [30]: `PPJoin` [33] first proposes a fixed-length prefix scheme where the l-prefix scheme takes the first $|R| - c + l$ elements of the set R as its prefix. `PPJoin` proves that two sets are similar only if their l-prefixes share at least l common elements. `AdaptJoin` further proposes an adaptive prefix scheme to improve the fixed length prefix scheme. It develops a cost model to select an appropriate prefix scheme for each set. It builds an incremental inverted index for all the elements with position information, i.e., the inverted list of an element consists of all the sets containing this element and its positions in the sets. For the set R with l-prefix scheme, `AdaptJoin` retrieves all the inverted lists of the elements in its prefix, scans those elements in the prefix of some sets using the position, outputs all the sets sharing at least l common elements in their prefixes as candidates, and verifies them. Nevertheless, its worst-case time complexity is still $O(n^2)$.

Note `ScanCount` and `DivideSkip` were original designated for search queries. We adapted them to do joins by conducting a search query for each set. For all the experiments, we varied c from 4 to 12 for DBLP and CLICK, 4 to 20 for ORKUT, and 2 to 6 for ADDRESS. The thresholds were 40% to 120% of the average set size on DBLP, 50% to 150% on CLICK, 5% to 25% on ORKUT, and 30% to 85% on ADDRESS, which are wide in relation to the average set size.

We implemented `All-Pairs` by ourselves and obtained the source code from the corresponding authors for the rest. All the methods were implemented in C++ and compiled using g++ 4.8.4 with -O3 flag. All experiments were conducted on a machine with Ubuntu 14.04 LTS, an Intel(R) Xeon(R) CPU E7-4830 @ 2.13GHz processor, and 256 GB memory.

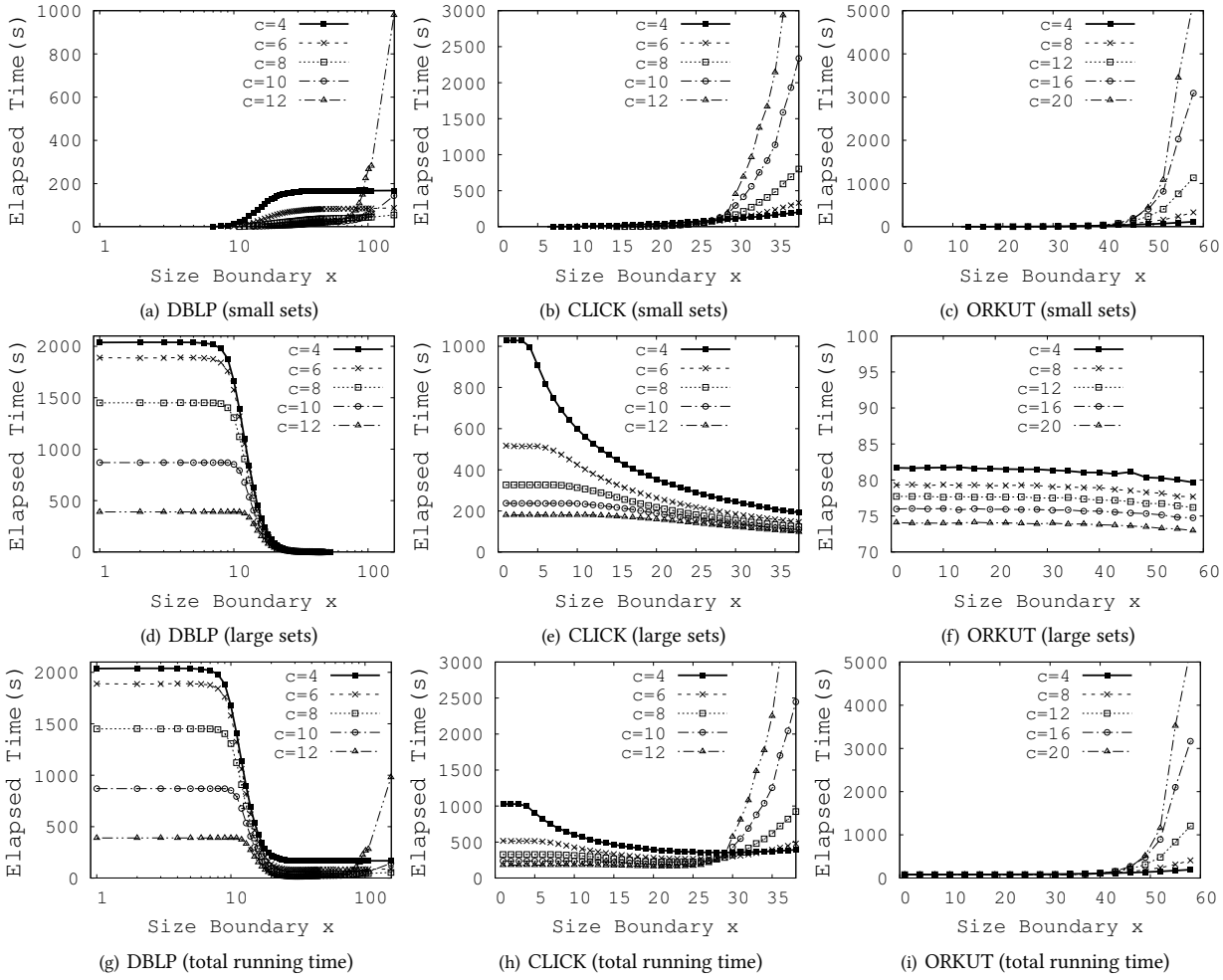
6.2 Evaluating The Heap-based Methods

The first set of experiments aimed to identify the best heap-based method for processing small sets. For this purpose, we used all the sets with sizes no larger than 30, 25, and 30 from DBLP, CLICK, and ORKUT to conduct the experiment, which results in 998618, 934203, and 359124 small sets respectively. As all the sets in ADDRESS are quite small, we used all of them in this set of experiments.

We implemented the following five methods: (1) `Naive`, which enumerates all c-subsets for each small set; (2) `HeapSkip`, which utilizes a min-heap to skip unique c-subsets; (3) `HeapDedup`, which utilizes a min-heap to skip both unique c-subsets and adjacent redundant c-subsets; (4) `BlockSkip`, which first blocks the c-subsets and then utilizes a min-heap for each block to skip unique c-subsets; (5) `BlockDedup`, which first blocks the c-subsets and then utilizes a min-heap for each block to skip both unique c-subsets and adjacent redundant c-subsets.

We first varied the threshold and reported the number of enumerated c-subsets (which is equal to the number of heap popping operations). Figure 6(a)-(d) gives the results. We observed that `BlockDedup` and `HeapDedup` enumerated the least number of c-subsets, and reduced that of `Naive` by up to 6 orders of magnitudes. For example, on ORKUT dataset when $c = 12$, the numbers of enumerated c-subsets for `Naive`, `HeapSkip`, `BlockSkip`, `HeapDedup`, and `BlockDedup` were respectively 2.2 trillion, 123 million, 122 million, 4.3 million, and 3.5 million. The reason behind the effectiveness of `BlockDedup` and `HeapDedup` is two-fold. First, they can skip all the adjacent redundant c-subsets. Second, their lazy insertion technique can skip more unique c-subsets than `HeapSkip` and `BlockSkip`. We can also see that `BlockDedup` and `BlockSkip` enumerated a little fewer c-subsets than `HeapDedup` and `HeapSkip`. This is because after blocking, some small sets could be directly dropped as they have less than c elements that are larger than the one used for blocking. For ADDRESS dataset, as the set sizes are quite small, the number of c-subsets for each set is limited (no larger than 35 for any $c \in [2, 6]$), which leads the gap between different

[10] A proof sketch is presented in Appendix C.



(a) DBLP (small sets) (b) CLICK (small sets) (c) ORKUT (small sets)

(d) DBLP (large sets) (e) CLICK (large sets) (f) ORKUT (large sets)

(g) DBLP (total running time) (h) CLICK (total running time) (i) ORKUT (total running time)

Figure 7: Evaluating the Size Boundary Selection Method

methods much smaller than those of the other datasets. Nevertheless, we still observed that the heap-based methods beat the naive method, and HeapDedup and BlockDedup enumerated less number of c-subsets than HeapSkip and BlockSkip.

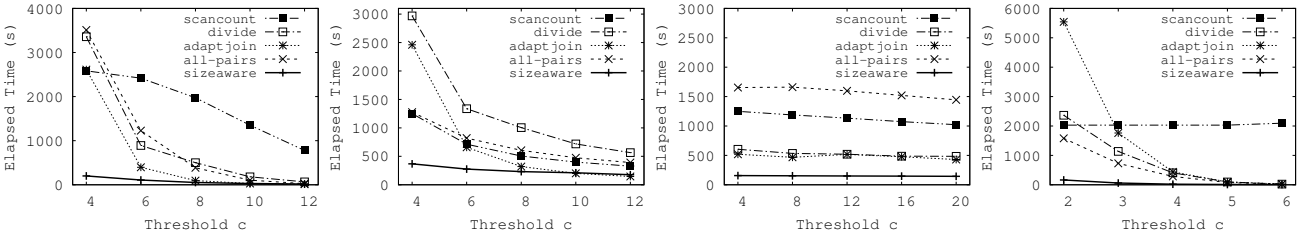
We also measured the total running time for the heap-based methods by varying the thresholds. The results are shown in Figure 6(e)-(h). We have the following observations. Firstly, BlockDedup and HeapDedup respectively outperformed BlockSkip and HeapSkip all the time as the former had less number of heap popping operations. Secondly, BlockDedup and BlockSkip respectively beat HeapDedup and HeapSkip in all the cases as the former had a smaller unit heap popping cost. Thirdly, BlockDedup consistently achieved the best performance as it not only required fewer popping operations but also had a smaller unit heap popping cost. The following experiments utilized BlockDedup as the designated method to process the small sets, due to its best overall efficiency. We also measured the elapsed time for Naive method. However, for DBLP, CLICK, and ORKUT, Naive reported the out-of-memory error after a long time (>1000s) in almost all the cases. Thus we only reported the results on ADDRESS dataset, as shown in Figure 6(h). We can see that on the only dataset that Naive can handle, BlockDedup still outperformed Naive by several times when $c \in [3, 6]$. However, Naive beat BlockDedup when $c = 2$. This is because the sets in ADDRESS are very small, which leads a small total number of c-subsets. In addition, the chance for BlockDedup to skip c-subsets decreased when c becomes small while BlockDedup needs more time to enumerate a c-subset than Naive as it used a heap to do so.

	by complexity		by our method			the best	
	x	time (sec)	x	time (sec)	accuracy	x	time (sec)
DBLP, $c = 4$	4	2042.5	30	174.5	112.6%	36	172
DBLP, $c = 6$	4	1894.9	34	85.99	80.9%	30	85.56
DBLP, $c = 8$	4	1455.6	32	38.41	98.2%	32	38.41
DBLP, $c = 10$	3	873.2	29	20.07	83.8%	31	19.71
DBLP, $c = 12$	3	392.6	29	11.04	112.7%	31	10.78
CLICK, $c = 4$	4	1000	30	358.12	89.1%	28	357.5
CLICK, $c = 6$	3	516.4	23	270.29	131.9%	24	269.4
CLICK, $c = 8$	3	329.1	24	224.44	80.1%	23	222
CLICK, $c = 10$	2	238.2	21	193.4	91.6%	21	193.4
CLICK, $c = 12$	2	182.3	25	182.96	78.6%	21	162.7
ORKUT, $c = 4$	5	149.4	8	149.5	97.0%	4	149.4
ORKUT, $c = 8$	3	146.9	11	146.8	115.8%	13	146.7
ORKUT, $c = 12$	2	145.1	15	144.8	97.3%	16	144.7
ORKUT, $c = 16$	2	142.9	18	142.7	84.9%	13	142.1
ORKUT, $c = 20$	2	139.7	22	139.4	91.6%	22	139.4

Table 3: The selected size boundaries

6.3 Evaluating The Size Boundary Selection

The experiments in this subsection focused on the behavior of our size-boundary selection method. Note for ADDRESS dataset, as all



(a) DBLP (b) CLICK (c) ORKUT (d) ADDRESS

Figure 8: Comparison with Existing Methods: Overlap Threshold

the sets have exactly the same size, our size boundary selection method classified all of them as small sets. The experiment results were less interesting for ADDRESS than the other three datasets and thus we omitted them here. We first enumerated a number of size boundaries and evaluated the processing time for small sets and large sets in the size-aware algorithm. Figures 7(a)-(c) present the processing time on the small sets, Figures 7(d)-(f) show the results for the large ones, and Figures 7(g)-(i) give the total running time. As the size boundary increased, the cost reduction for the large sets was considerable initially but then became insignificant later; while for the small sets, the cost growth was slow at the beginning, and then dramatically accelerated. For example, on DBLP dataset when $c = 12$, on size boundary 7, 17, 27, and 37, the elapsed time for small sets was 0.03s, 0.98s, 4.4s, and 11.1s respectively and 390s, 111s, 7.8s, and 2.2s for large sets. This is consistent with our time complexity analysis in Section 3.2. Due to this tradeoff between small and large sets, the overall cost of the size-aware algorithm first decreased and then increased. For example, on DBLP dataset when $c = 12$, the elapsed time for size boundary 12, 21, 31, 61, and 101 was respectively 377s, 35s, 10.8s, 25.4s, and 270s.

Table 3 shows the size boundaries that (i) were produced by the time complexity analysis, (ii) were chosen by our size-boundary selection method, and (iii) actually gave the best performance. We also reported the running time under each size boundary. We can see that our size boundary selection method was quite effective, and picked fairly good values that were close to the optimal ones. For example, on DBLP when $c = 8$, our size boundary selection method selected $x = 32$ as the boundary which achieved the optimal performance (38.41s) among all the enumerated boundaries. However, the time complexity analysis suggested $x = 4$ as the boundary which led a much worse running time of 1455s. This evidenced that the cost model in our method is accurate. Note the sixth column gives the estimation accuracy for the small sets, which was the ratio of the estimated costs to the real costs for processing small sets. We can see the cost estimation is accurate. The cost estimation for large sets is always accurate as it does not use sampling techniques.

6.4 Comparison with Existing Methods: Overlap Threshold

We compared our size-aware algorithm with four existing methods `DivideSkip`, `All-Pairs`, `AdaptJoin`, and `ScanCount` by varying the threshold c. Figure 8 reports the total running time as a function of c. Our size-aware method always achieved the best performance and outperformed the others by up to an order of magnitude. For example, on DBLP dataset when $c = 4$, the elapsed time for `ScanCount`, `DivideSkip`, `AdaptJoin`, `All-Pairs`, and `SizeAware` was respectively 2585s, 3358s, 2612s, 3509s, and 161s. The main reason for this is the existing methods spent considerable time scanning the element inverted lists, while our size-aware method avoided this by separately processing the small sets and the large sets. Moreover, the size boundary selection method can select a good size boundary for the size-aware algorithm. In addition, with the increase of the threshold c, the running time decreased because there were fewer answers and fewer sets with sizes no smaller than c. We have the same observation on ADDRESS, because `SizeAware` generates the results directly from the c-subsets, whose total number is small as the sets in ADDRESS are very small. However, scanning the element inverted index took a long time for existing methods. Moreover, when all the sets have the same size, the overlap similarity is equivalent to Jaccard similarity (see details in Appendix D). We compared `SizeAware` with some additional existing methods for set similarity join with Jaccard constraint on ADDRESS dataset in Appendix E.

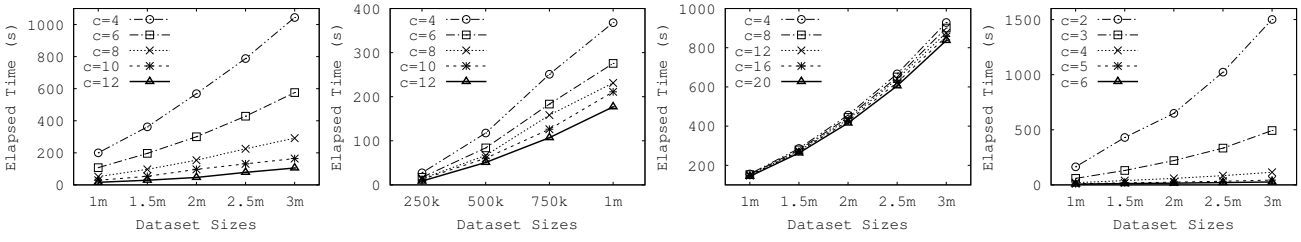
6.5 Comparison with Existing Methods: Scalability

The last set of experiments studied the scalability of our method. We varied the dataset sizes from 1 million to 3 million for DBLP dataset, 250,000 to around 1 million for CLICK, 1 million to 3 million for ORKUT, and 1 million to 3 million for ADDRESS dataset. The elapsed time of `SizeAware` under different thresholds is reported in Figure 9. We can see that our methods achieved sub-quadratic scalability, which is consistent with our time complexity analysis. For example, on DBLP dataset, when the threshold $c = 4$, the elapsed time for 1 million, 1.5 million, 2 million, 2.5 million, and 3 million sets was respectively 200s, 362s, 569s, 788s, and 1044s. This is because `SizeAware` processes small sets and large sets separately using two methods that are scalable to sets with different sizes. In addition, the size boundary selection method can properly dichotomize the input dataset. We also evaluated the scalability of `SizeAware` in $\mathcal{R}$-$\mathcal{S}$ join case and report the results in Appendix E.

We also compared our scalability with the existing work. Figures 10 gives the results. We varied the dataset sizes and reported the elapsed time for different methods under specific thresholds. We can see that our method achieved the best scalability. For example, on ORKUT dataset, under the threshold $c = 12$, when there were 1 million sets, the elapsed time for `ScanCount`, `DivideSkip`, `AdaptJoin`, `All-Pairs`, and `SizeAware` was respectively 1130s, 520s, 1600s, 520s, and 150s; while it was 9885s, 4585s, 15500s, 4400s, and 875s when there were 3 million sets. The elapsed time increased 8.75, 8.82, 9.69, 8.46, and 5.83 times when the dataset size increased 3 times. This is because all existing methods had a quadratic worst-case time complexity and their filtering techniques had little effect when the threshold was relatively small compared to the set sizes.

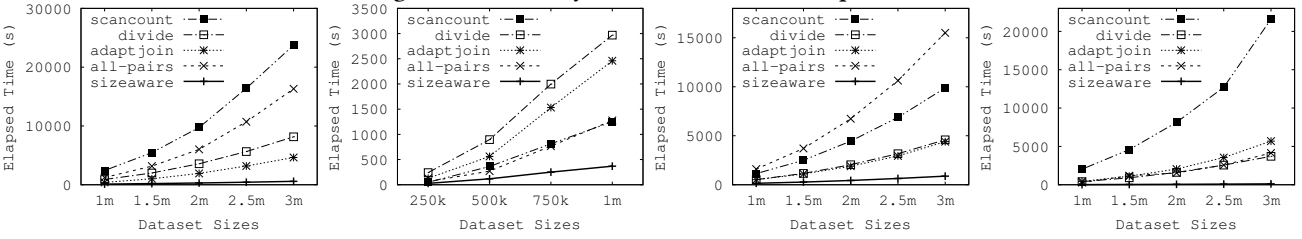
7 RELATED WORK

Set Similarity Join and Search with Overlap Constraints. Broder et al. [5] proposed to build an inverted index for the elements and enumerate every set pair in each inverted list to find the set pairs



(a) DBLP (b) CLICK (c) ORKUT (d) ADDRESS

Figure 9: Scalability under Different Overlap Thresholds



(a) DBLP (c=6) (b) CLICK (c=4) (c) ORKUT (c=12) (d) ADDRESS (c=4)

Figure 10: Comparison with Existing Methods: Scalability

with enough overlap. This is different from our method for small sets where we resort to element subsets with size c. Sarawagi et al. [25] proposed a threshold sensitive list merge algorithm for set similarity join. Li et al. [13] improved the list merge algorithm and adapted it to set similarity search. Chaudhuri et al. [7] proposed the prefix filter technique and used it as a primitive operator in a database system for similarity join. Bayardo et al. [3] proposed a similar approach for solving the same problem under in-memory setting. Wang et al. [30] improved the prefix filter by proposing a cost model to create adaptive prefix filters. Teflioudi et al. [29] studied the inner product join problem which takes vectors instead of sets as the input and utilizes the vector inner product in the join constraint (instead of the overlap as in our problem).

Similarity Join and Search with Other Constraints. Similarity join and search with other constraints, such as Jaccard, Cosine, Hamming, Edit Distance and Containment, are extensively studied [2, 3, 7–9, 15, 19, 31, 33]. Xiao et al. [33] proposed `PPJoin` and `PPJoin+` for set similarity join with Jaccard, Cosine and Dice constraints which improve the prefix filter by considering the element positions. Bouros et al. [4] designed `GroupJoin` to group the same prefixes to share computation. Wang et al. [31] developed `SKJ` which can skip scanning a part of the inverted lists. Mann et al. [17] proposed `PEL` to improved the length filter using the position information. Deng et al. [9, 32] proposed a partition-based method. Arasu et al. [2] developed a partition-and-enumeration method for the set similarity join with Jaccard and Hamming constraints. All of them use the filter-and-refine framework [9]. Melnik et al. [19] proposed partition-based algorithms for set containment join. Note our work is different from the set containment join works [24, 34] as they aim to find set pairs with the containment relationship. Li et al. [14] proposed a partition-based method for string similarity join with the edit distance constraint. Deng et al. [8] proposed a pivotal prefix filter for string similarity search. [12, 18] conducted experimental evaluations on the similarity join problem. We discuss more details about the relationship between the set similarity join with overlap constraint and the other constraints and experimentally compare `SizeAware` with them in Appendixes D and E.

Approximate Similarity Join and Search Algorithms. There is a rich literature [1, 21, 22, 26–28] on approximate algorithms for set similarity join and search. Most of them are related to locality sensitive hashing (LSH) [10, 11]. The idea behind LSH is to partition the input sets into buckets such that the more similar two sets are, the higher probability they are hashed to the same bucket. Pagh [22] proposed the LSH for hamming distance without false negatives. The traditional LSH cannot support the non-metric space distance function. To address this issue, Shrivastava et al. [27] proposed an asymmetric LSH which pre-processes the vectors by asymmetric transformation to make them fit in the classic LSH technique. However, it is non-trivial to extend these techniques to support the threshold-based overlap set similarity join query, because the overlap between two similar sets can be vanishingly small compared to the size of the sets and the tricks like picking a random element and expecting it to be in both sets do not work.

8 CONCLUSION

In this paper, we study the set similarity join problem with overlap constraints. We propose a size-aware algorithm with the time complexity of $O(n^{2-\frac{1}{c}}k^{\frac{1}{2c}})$ where n is the total size of all the sets and k is the number of results. We divide all the sets into small sets and large sets and process them separately. For the small sets, we enumerate all their c-subsets and take any set pair sharing at least one c-subset as a result. To avoid enumerating unnecessary c-subsets, we develop a heap-based method to avoid the unique c-subsets that cannot generate any result and the redundant c-subsets that only generate duplicate results. We propose to block the c-subsets to reduce the heap size and the heap-adjusting cost. We design an effective method to select an appropriate size boundary. Experimental results show that our algorithm outperforms state-of-the-art studies by up to an order of magnitude.

Acknowledgment: The research of Yufei Tao was partially supported by a direct grant (Project Number: 4055079) from CUHK and by a Faculty Research Award from Google. Guoliang Li was supported by the 973 Program of China (2015CB358700), NSF of China (61632016,61472198,61521002,61661166012), and TAL education.

REFERENCES

[1] T. D. Ahle, R. Pagh, I. P. Razenshteyn, and F. Silvestri. On the complexity of inner product similarity join. In *PODS*, pages 151–164, 2016.
[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.
[4] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.
[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.
[6] J. A. Bullinaria and J. P. Levy. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods*, 39(3):510–526, Aug 2007.
[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.
[8] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.
[9] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.
[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.
[11] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(1):321–350, 2012.
[12] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.
[13] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[14] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9, 2013.
[15] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.
[16] K. Lund and C. Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, 28(2):203–208, Jun 1996.
[17] W. Mann and N. Augsten. PEL: position-enhanced length filter for set similarity joins. In *GVD*, pages 89–94, 2014.
[18] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.
[19] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28:56–99, 2003.
[20] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.
[21] B. Neyshabur and N. Srebro. On symmetric and asymmetric lshs for inner product search. In *ICML*, pages 1926–1934, 2015.
[22] R. Pagh. Locality-sensitive hashing without false negatives. In *SODA*, pages 1–9, 2016.
[23] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.
[24] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, 2000.
[25] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, 2004.
[26] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.
[27] A. Shrivastava and P. Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NIPS*, pages 2321–2329, 2014.
[28] A. Shrivastava and P. Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *WWW*, pages 981–991, 2015.
[29] C. Teflioudi, R. Gemulla, and O. Mykytiuk. LEMP: fast retrieval of large entries in a matrix product. In *SIGMOD*, pages 107–122, 2015.
[30] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.
[31] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.
[32] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.
[33] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.
[34] J. Yang, W. Zhang, S. Yang, Y. Zhang, and X. Lin. Tt-join: Efficient set containment join. In *ICDE*, pages 509–520, 2017.
[35] Y. Zhang, G. Lai, M. Zhang, Y. Zhang, Y. Liu, and S. Ma. Explicit factor models for explainable recommendation based on phrase-level sentiment analysis. In *SIGIR*, pages 83–92, 2014.

A BINARY SEARCHING FOR C-SUBSETS

Given the r_c^{min} from R and the r_c^{top}, when $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ we require to find the smallest c-subset in R that is not smaller than r_c^{top}. We can achieve this by binary searching R. Suppose that $R = \{e'_1, e'_2, \cdots, e'_{|R|}\}$ and $\mathsf{r}_c^{top} = e_1 e_2 \cdots e_c$ where $e_i < e_j$ and $e'_i < e'_j$ for any $i < j$. For each element e_i in increasing order, we binary search the smallest element e'_{a_i} in R that is not smaller than e_i until we first meet $e_i \neq e'_{a_i}$. Then we reinsert the c-subset $e'_{a_1} e'_{a_2} \cdots e'_{a_i} e'_{a_i+1} \cdots e'_{a_i+c-i}$ into the heap where $e'_{a_1} = e_1, e'_{a_2} = e_2, \cdots, e'_{a_{i-1}} = e_{i-1}$ and $e'_{a_i} \neq e_i$ and $e'_{a_i+1} \cdots e'_{a_i+c-1}$ are the elements right after e'_{a_i} in R.

B ADAPTATION FOR R-S JOIN

In this section, we extend our theoretical results and techniques to the $\mathcal{R}$-$\mathcal{S}$ join case (where $\mathcal{R} \neq \mathcal{S}$).

The size-aware algorithm for the $\mathcal{R}$-$\mathcal{S}$ join. Given two collections of sets $\mathcal{R}$ and $\mathcal{S}$ and a threshold c, the size-aware algorithm divides all the sets into large sets $\mathcal{R}_l$ and $\mathcal{S}_l$ and small sets $\mathcal{R}_s$ and $\mathcal{S}_s$ with the size boundary x and processes them separately. For each large set $R \in \mathcal{R}_l$ (or $S \in \mathcal{S}_l$), it compares R (or S) with every set in $\mathcal{S}$ (or $\mathcal{R}$). As there are totally at most $\frac{n}{x}$ large sets where n is the total size of all the sets, the time complexity of processing the large sets is $O(\frac{n^2}{x})$. For each small set, the size-aware algorithm enumerates all its c-subsets. As the size of a small set is no larger than x and the number of c-subsets for a small set $R \in \mathcal{R}_s$ (or $S \in \mathcal{S}_s$) is within $|R|^c$ (or $|S|^c$), the number of all c-subsets cannot exceed $x^{c-1}n$ and the time complexity of enumerating c-subsets is $O(x^{c-1}n)$. Next it generates all the results from the c-subset inverted index. Suppose that the c-subset inverted lists generated by $\mathcal{R}_s$ and $\mathcal{S}_s$ are respectively $\mathcal{L}_1, \mathcal{L}_2, \ldots \mathcal{L}_l$ and $\mathcal{L}'_1, \mathcal{L}'_2 \ldots \mathcal{L}'_l$ where $\mathcal{L}_i$ and $\mathcal{L}'_i$ associated with the same c-subset. The time complexity of generating the results is

$$O(\sum_{i=1}^{l} |\mathcal{L}_i| \times |\mathcal{L}'_i|).$$

As the number of results generated from any inverted list cannot exceed k, we have

$$\min(\frac{|\mathcal{L}_i|(|\mathcal{L}_i|-1)}{2}, \frac{|\mathcal{L}'_i|(|\mathcal{L}'_i|-1)}{2}) \leq k.$$

It thus follows that $\min(|\mathcal{L}_i|, |\mathcal{L}'_i|) = O(\sqrt{k})$. Moreover, as the total size of all inverted lists, which is exactly the number of all c-subsets, cannot be larger than $x^{c-1}n$, we have

$$\sum_{i=1}^{l} \max(|\mathcal{L}_i|, |\mathcal{L}'_i|) \leq \sum_{i=1}^{l} (|\mathcal{L}_i| + |\mathcal{L}'_i|) \leq x^{c-1}n.$$

Thus the time complexity of generating the results is

$$\begin{aligned} O(\sum_{i=1}^{l} |\mathcal{L}_i| \times |\mathcal{L}'_i|) &= O\Big(\sum_{i=1}^{l} \min(|\mathcal{L}_i|, |\mathcal{L}'_i|) \times \max(|\mathcal{L}_i|, |\mathcal{L}'_i|)\Big) \\ &= O\Big(\sqrt{k} \sum_{i=1}^{l} \max(|\mathcal{L}_i|, |\mathcal{L}'_i|)\Big) = O(x^{c-1}n\sqrt{k}). \end{aligned}$$

Algorithm 6: SizeAwareRSJoin

Input: $\mathcal{R}$: a dataset; $\mathcal{S}$: another dataset; c : a threshold.
Output: $\mathcal{A} = \{\langle R, S\rangle \big| |R \cap S| \geq c, R \in \mathcal{R}, S \in \mathcal{S}\}$.

1 x = GetSizeBoundary($\mathcal{R}, \mathcal{S}, c$);
2 divide $\mathcal{R}$ and $\mathcal{S}$ into small sets $\mathcal{R}_s$ and $\mathcal{S}_s$ and large sets $\mathcal{R}_l$ and $\mathcal{S}_l$ by the size boundary x;
3 Using ScanCount to find all the similar set pairs in $\mathcal{R}_l \times \mathcal{S}$ and $\mathcal{S}_l \times \mathcal{R}$ and add them into $\mathcal{A}$;
4 $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle$ = BlockDedup($\mathcal{R}_s, \mathcal{S}_s, c$);
5 **foreach** r_c *s.t.* $\mathcal{L}_{slim}[\mathsf{r}_c] \neq \phi$ & $\mathcal{L}'_{slim}[\mathsf{r}_c] \neq \phi$ **do**
6 add every set pair in $\mathcal{L}_{slim}[\mathsf{r}_c] \times \mathcal{L}'_{slim}[\mathsf{r}_c]$ into $\mathcal{A}$;
7 **return** $\mathcal{A}$;

Algorithm 7: HeapDedup($\mathcal{R}_s$, $\mathcal{S}_s$, c)

Input: $\mathcal{R}_s$: a collection of small sets; c: a threshold; $\mathcal{S}_s$: another collection of small sets.
Output: $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle$: slimmed inverted indexes.

1 Insert all the min-subsets in $\mathcal{R}_s$ and $\mathcal{S}_s$ to heaps $\mathcal{H}$ and $\mathcal{H}'$;
2 Pop $\mathcal{H}$ and $\mathcal{H}'$ to get the smallest c-subsets r_c^{min} and s_c^{min} ;
3 **while** *neither* $\mathcal{H}$ *nor* $\mathcal{H}'$ *is empty* **do**
4 Suppose r_c^{min} and s_c^{min} are from R and S respectively;
5 **if** $\mathsf{r}_c^{min} > \mathsf{s}_c^{min}$ **then**
6 append S to $\mathcal{L}'_{slim}[\mathsf{s}_c^{min}]$, binary search S for the first c-subset that is no smaller than r_c^{min}, reinsert it into $\mathcal{H}'$, and pop $\mathcal{H}'$ to get the next s_c^{min};
7 **else if** $\mathsf{r}_c^{min} < \mathsf{s}_c^{min}$ **then**
8 append R to $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$, binary search R for the first c-subset that is no smaller than s_c^{min}, reinsert it into $\mathcal{H}$, and pop $\mathcal{H}$ to get the next r_c^{min};
9 **else if** $\mathsf{r}_c^{min} = \mathsf{s}_c^{min}$ **then**
10 **while** $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ **do**
11 append R to $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$, pop $\mathcal{H}$ to get next r_c^{min};
12 **while** $\mathsf{s}_c^{min} \neq \mathsf{s}_c^{top}$ **do**
13 append S to $\mathcal{L}'_{slim}[\mathsf{s}_c^{min}]$, pop $\mathcal{H}'$ to get next s_c^{min};
14 **return** $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle$;

The rest is the same as the self-join case and the time complexity of the size-aware algorithm is still

$$O(n^{2-\frac{1}{c}} k^{\frac{1}{2c}}) = o(n^2) + O(k).$$

The pseudo codes of the size-aware algorithm for the $\mathcal{R}$-$\mathcal{S}$ join case is shown in Algorithm 6. It first utilizes the size boundary selection method to choose a size boundary x (Line 1). Then it divides the two datasets using x (Line 2). It uses ScanCount to find the results in $\mathcal{R}_l \times \mathcal{S}$ and $\mathcal{S}_l \times \mathcal{R}$ and add them to the result set $\mathcal{A}$ (Lines 3). Next it utilizes the BlockDedup method to generate the slimmed inverted indexes $\mathcal{L}_{slim}$ and $\mathcal{L}'_{slim}$ for $\mathcal{R}_s$ and $\mathcal{S}_s$ respectively (Line 4). Finally, for each c-subset r_c, it adds every pair in $\mathcal{L}_{slim}[\mathsf{r}_c] \times \mathcal{L}'_{slim}[\mathsf{r}_c]$ to $\mathcal{A}$ and returns $\mathcal{A}$ (Lines 5 to 7).

The HeapSkip method for the $\mathcal{R}$-$\mathcal{S}$ join. Suppose $\mathcal{L}$ and $\mathcal{L}'$ are the c-subset inverted indexes constructed from $\mathcal{R}_s$ and $\mathcal{S}_s$. For any c-subset r_c, if $\mathcal{L}[\mathsf{r}_c] = \phi$ or $\mathcal{L}'[\mathsf{r}_c] = \phi$, it cannot generate any result and we call it a unique c-subset. To skip the unique c-subsets, we fix a global order for all the c-subsets and access the c-subsets in each set in order. We build two min-heaps $\mathcal{H}$ and $\mathcal{H}'$ to maintain the min-subsets of the sets in $\mathcal{R}_s$ and $\mathcal{S}_s$ respectively. We pop $\mathcal{H}$ and $\mathcal{H}'$ and get the smallest min-subsets r_c^{min} and s_c^{min} in $\mathcal{H}$ and $\mathcal{H}'$. Suppose that they come from R and S respectively. We compare r_c^{min} with s_c^{min}. If $\mathsf{r}_c^{min} = \mathsf{s}_c^{min}$, we first append R to $\mathcal{L}[\mathsf{r}_c^{min}]$ and S to $\mathcal{L}'[\mathsf{s}_c^{min}]$ and then reinsert the next min-subsets in R and S to $\mathcal{H}$ and $\mathcal{H}'$ respectively. If $\mathsf{r}_c^{min} > \mathsf{s}_c^{min}$, we first append S to $\mathcal{L}'[\mathsf{s}_c^{min}]$ and then reinsert the smallest c-subset that is no smaller than r_c^{min} in S to $\mathcal{H}'$ by binary searching. Otherwise $\mathsf{s}_c^{min} > \mathsf{r}_c^{min}$, we first append R to $\mathcal{L}[\mathsf{r}_c^{min}]$ and then reinsert the smallest c-subset that is no smaller than s_c^{min} in R to $\mathcal{H}$ by binary searching. We repeat this until $\mathcal{H}$ or $\mathcal{H}'$ is empty.

The HeapDedup method for the $\mathcal{R}$-$\mathcal{S}$ join. If the two inverted lists $\mathcal{L}[\mathsf{r}_c]$ and $\mathcal{L}'[\mathsf{r}_c]$ of a c-subset r_c are sub-lists of those of another c-subset r'_c, i.e., $\mathcal{L}[\mathsf{r}_c] \subseteq \mathcal{L}[\mathsf{r}'_c]$ and $\mathcal{L}'[\mathsf{r}_c] \subseteq \mathcal{L}'[\mathsf{r}'_c]$, r_c can only generate duplicate results and we call it a redundant c-subset. To skip the adjacent redundant c-subsets, we delay the reinsertion of min-subsets to the heaps when $\mathsf{r}_c^{min} = \mathsf{s}_c^{min}$. More specifically, when $\mathsf{r}_c^{min} = \mathsf{s}_c^{min}$, we keep popping $\mathcal{H}$ ($\mathcal{H}'$) until $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ ($\mathsf{s}_c^{min} \neq \mathsf{s}_c^{top}$) where r_c^{top} (s_c^{top}) is the c-subset currently tops $\mathcal{H}$ ($\mathcal{H}'$). Then for each set in $\mathcal{L}[\mathsf{r}_c^{min}]$ ($\mathcal{L}'[\mathsf{s}_c^{min}]$), we reinsert the smallest c-subsets in it that is no smaller than s_c^{top} (r_c^{top}) to $\mathcal{H}$ ($\mathcal{H}'$) by binary searching. This is because the c-subsets between r_c^{min} and s_c^{top} in $\mathcal{R}$ and between s_c^{min} and r_c^{top} in $\mathcal{S}$ do not appear in the other sets except those in $\mathcal{L}[\mathsf{r}_c^{min}]$ and $\mathcal{L}'[\mathsf{s}_c^{min}]$ and must be redundant c-subsets. The rest is the same as the self-join case.

Algorithm 8: BlockDedup($\mathcal{R}_s$, $\mathcal{S}_s$, c)

Input: $\mathcal{R}_s$: a collection of small sets; c: a threshold; $\mathcal{S}_s$: another collection of small sets.
Output: $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle$: slimmed inverted indexes.

1 Fix a global order for all the elements in $\mathcal{R}_s$ and $\mathcal{S}_s$;
2 Build element inverted indexes $\mathcal{I}$ and $\mathcal{I}'$ for $\mathcal{R}_s$ and $\mathcal{S}_s$;
3 **foreach** *element* e *s.t.* $\mathcal{I}[e] \neq \phi$ *and* $\mathcal{I}'[e] \neq \phi$ **do**
4 $\mathcal{R}_{tmp}$ = *sets in* $\mathcal{I}[e]$ *with elements* $\leq e$ removed;
5 $\mathcal{S}_{tmp}$ = *sets in* $\mathcal{I}'[e]$ *with elements* $\leq e$ removed;
6 $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle = \langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle \cup$ HeapDedup($\mathcal{R}_{tmp}$, $\mathcal{S}_{tmp}$, $c-1$);
7 **return** $\langle \mathcal{L}_{slim}, \mathcal{L}'_{slim}\rangle$;

The pseudo-code of the HeapDedup method for $\mathcal{R}$-$\mathcal{S}$ join is shown in Algorithm 7. It takes two collections of small sets $\mathcal{R}_s$ and $\mathcal{S}_s$ as input and outputs two slimmed inverted indexes $\mathcal{L}_{slim}$ and $\mathcal{L}'_{slim}$ for $\mathcal{R}_s$ and $\mathcal{S}_s$ respectively. It first initializes two min-heaps $\mathcal{H}$ and $\mathcal{H}'$ for $\mathcal{R}_s$ and $\mathcal{S}$ and pops out the smallest min-subsets r_c^{min} and s_c^{min} from $\mathcal{H}$ and $\mathcal{H}'$ (Lines 1 to 2). Suppose that r_c^{min} and s_c^{min} are come from R and S respectively. It keeps comparing r_c^{min} and s_c^{min} until either $\mathcal{H}$ or $\mathcal{H}'$ is empty (Line 3). If $\mathsf{r}_c^{min} > \mathsf{s}_c^{min}$, it first appends S to $\mathcal{L}'_{slim}[\mathsf{s}_c^{min}]$, then binary searches S for the first c-subset that is no smaller than r_c^{min}, next reinserts it to $\mathcal{H}'$, and finally pops $\mathcal{H}'$ to get the next s_c^{min} (Line 6). If $\mathsf{r}_c^{min} < \mathsf{s}_c^{min}$, it first appends R to $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$, then binary searches R for the first c-subset that is no smaller than s_c^{min}, next reinserts it to $\mathcal{H}$, and finally pops $\mathcal{H}$ to get the next r_c^{min} (Line 8). If $\mathsf{r}_c^{min} = \mathsf{s}_c^{min}$, it keeps popping $\mathcal{H}$

until $\mathsf{r}_c^{min} \neq \mathsf{r}_c^{top}$ and builds the inverted list $\mathcal{L}_{slim}[\mathsf{r}_c^{min}]$ (Line 11). Similarly it keeps popping $\mathcal{H}'$ until $\mathsf{s}_c^{min} \neq \mathsf{s}_c^{top}$ and builds the inverted list $\mathcal{L}'_{slim}[\mathsf{s}_c^{min}]$ (Line 13). In this way, it can construct two slimmed inverted indexes.

The blocking-based methods for the $\mathcal{R}$-$\mathcal{S}$ join. For the $\mathcal{R}$-$\mathcal{S}$ join case, we still block all the c-subsets based on the smallest elements. We build two element inverted indexes $\mathcal{I}$ and $\mathcal{I}'$ for the two datasets. Then for each element e, if $\mathcal{I}[e] \neq \phi$ and $\mathcal{I}'[e] \neq \phi$, we can independently apply the heap-based methods on the sets in $\mathcal{I}[e]$ and $\mathcal{I}'[e]$ by only inserting those c-subsets with the smallest element as e to the heaps.

The pseudo-code of the `BlockDedup` method for $\mathcal{R}$-$\mathcal{S}$ join is shown in Algorithm 7. It first fixes a global order for all the elements and builds two inverted indexes $\mathcal{I}$ and $\mathcal{I}'$ for the elements in $\mathcal{R}_s$ and $\mathcal{S}_s$ respectively (Lines 1 to 2). Then for each element e such that $\mathcal{I}[e] \neq \phi$ and $\mathcal{I}'[e] \neq \phi$, it builds two temporary collections of sets $\mathcal{R}_{tmp}$ and $\mathcal{S}_{tmp}$ by removing the elements no larger than e in $\mathcal{I}[e]$ and $\mathcal{I}'[e]$ respectively (Lines 4 to 5). It utilizes the `HeapDedup` method to construct the parts of the slimmed inverted indexes for the two temporary sets with the threshold $c-1$ (Line 6). Finally it can get two slimmed inverted indexes and return them (Line 7).

The boundary size selection method for the $\mathcal{R}$-$\mathcal{S}$ join. The boundary size selection method for the $\mathcal{R}$-$\mathcal{S}$ join is basically the same as that for the self-join case. The time cost for the large set is proportional to $\sum_{R\in\mathcal{R}_l}\sum_{e\in R}|\mathcal{I}'[e]| + \sum_{S\in\mathcal{S}_l}\sum_{e\in S}|\mathcal{I}[e]|$. We randomly sample small set pairs from $\mathcal{R}_s \times \mathcal{S}_s$ to estimate the result generation cost. We randomly sample blocks to estimate the heap adjusting cost and the binary searching cost. We have the same observation as that of the self-join case on the trends of the time complexities of small sets and large sets with the increase of the size boundary x. Thus the cost model is all the same. We first set the size boundary x as the smallest set size in both datasets or c, whichever is larger and try to increase x by 1 each time. We use the increasing of the time cost for small sets as the cost and the decreasing of the time cost for large sets as the benefit. We stop increasing x when the benefit is smaller than the cost.

C THE TIME COMPLEXITY OF PREFIX FILTER

Here is an example to show that the prefix filter has a worst case time complexity of $O(n^2)$. Suppose there is a constant number p of distinct elements in the sets, all the elements have the same frequency $\frac{n}{p}$, and the sizes of all the sets are larger than c. There exists at least one element (the first element in the global order used by `All-Pairs`) whose corresponding inverted list has a length of $\frac{n}{p}$ and is scanned $\frac{n}{p}$ times. Thus the complexity is $O(\frac{n}{p}\times\frac{n}{p}) = O(n^2)$.

D RELATIONSHIP WITH THE OTHER SIMILARITY FUNCTIONS

Just like all the similarity problems, set similarity join can also be studied under other functions, such as Jaccard similarity, Cosine similarity, Dice similarity, edit distance, and the normalized overlap similarity. Every metric has its pros and cons, such that no metric serves as a one-size-fits-all approach that cures all the issues coming up in practice. For example, in some applications, the set sizes may differ a lot, in which case most metrics will give low similarities to the "lop-sided" set pairs (where one set out-sizes the other significantly), whereas the overlap similarity is known to be much less sensitive to such an issue.

Nevertheless, when all the sets in the given dataset have the same sizes (e.g., the sets in ADDRESS), the overlap similarity can be transformed to Jaccard similarity, Cosine similarity, Dice similarity, and the normalized overlap similarity and vice versa. For example, suppose all the set sizes are m. Then the set pairs with overlap similarity no smaller than c are exactly the set pairs with Jaccard similarity no smaller than $\frac{c}{2m-c}$. This is because for any sets r and s in the given dataset, we have

$$\texttt{Jaccard}(r,s) = \frac{|r\cap s|}{|r\cup s|} = \frac{|r\cap s|}{|r|+|s|-|r\cap s|} = \frac{|r\cap s|}{2m-|r\cap s|}$$

and thus $|r\cap s| \geq c$ iff $\texttt{Jaccard}(r,s) \geq \frac{c}{2m-c}$. In this case, `SizeAware` and the methods for set similarity joins under the other similarity functions can be used to solve each other's problems. We compared `SizeAware` with five existing methods for set similarity joins under Jaccard similarity constraint on ADDRESS in Appendix E.

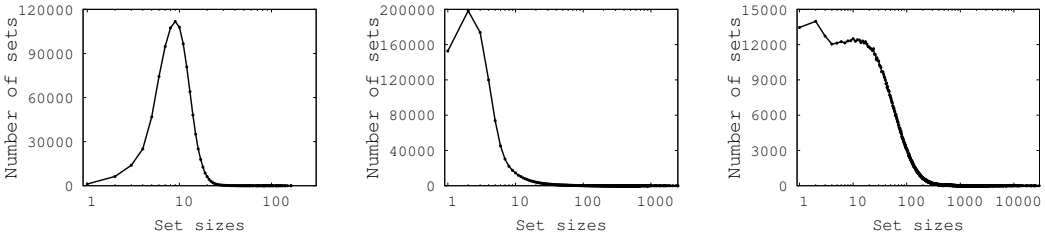
In general, when the set sizes are not all the same in the given dataset, all the alternative similarity functions mentioned earlier can actually be transformed to overlap similarity (but not vice versa), after which our `SizeAware` algorithm can be used to solve the set similarity join under those metrics as well. The transformation can be achieved using an existing technique called `AdaptJoin` [30]. For each set R, given a constant l, the fixed-length prefix schema of `AdaptJoin` takes the first $|R| - f(|R|) + l$ elements as prefixes, and guarantees that two sets are similar only if their prefixes share at least l elements, where $f(|R|)$ is a function that depends on the similarity metric. By taking all the l-prefixes as input and setting the overlap threshold $c = l$, our size-aware algorithm can efficiently identify the candidates for the set similarity join problem. The candidates can then be fed into a verification step to produce the final results. In practice, we can use the advance length filter [17], prefix filter [33], and position filter [31] to process the large sets and the sets R with $f(|R|) < c$. We can also use the estimation in `AdaptJoin` to select a good c.

The reverse transformation, on the other hand, does not always appear to be possible when the set sizes are not all the same. This fundamental nature of overlap similarity provides further motivation for our algorithmic study of this metric.

E MORE EXPERIMENTS

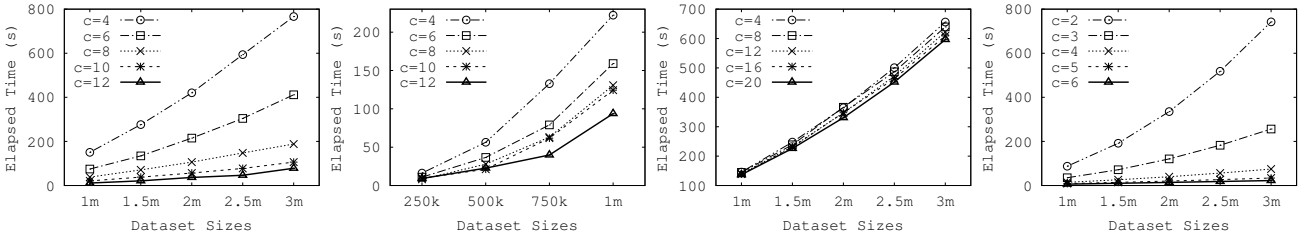
The Set Size Distributions. The set size distributions of DBLP, CLICK, and ORKUT are shown in Figure 11.

Memory Usage. We also compared the memory usage with existing methods and Table 4 gives the numbers. We can see that the memory usage of `SizeAware` is comparable to the existing approaches. In our heap-based methods, the smallest c-subsets r_c are first popped out from the heap and the inverted list $\mathcal{L}[\mathsf{r}_c]$ is constructed. We can enumerate the results in this inverted list and drop it immediately as it will never be used again later in the algorithm. Thus it only needs a small amount of memory to keep the results, the element inverted index $\mathcal{I}$, and the heap. However, the heap-based methods may generate duplicate results. In our implementation, we keep the whole slimmed inverted index for efficient result deduplication.



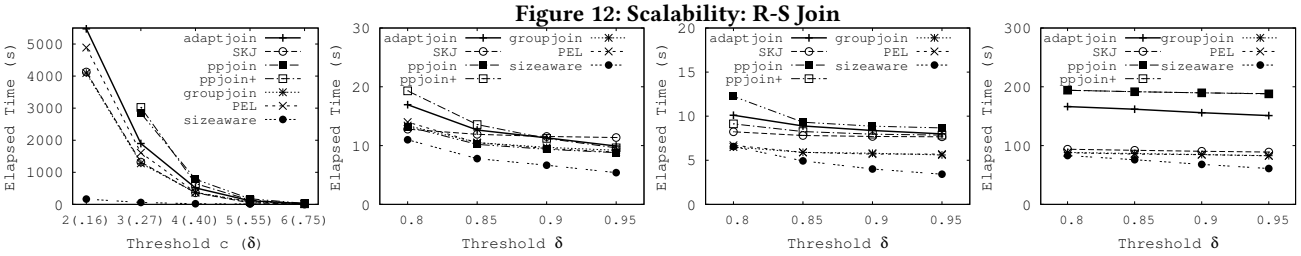
(a) DBLP (b) CLICK (c) ORKUT

Figure 11: Set Size Distributions



(a) DBLP (b) CLICK (c) ORKUT (d) ADDRESS

Figure 12: Scalability: R-S Join



(a) ADDRESS (b) DBLP (c) CLICK (d) ORKUT

Figure 13: Comparison with Existing Methods for Jaccard Set Similarity Join

	ScanCount	DivideSkip	AdaptJoin	AllPair	SizeAware
DBLP, $c = 8$	180 MB	199 MB	308 MB	309 MB	579 MB
CLICK, $c = 8$	142 MB	2251 MB	304 MB	241 MB	940 MB
ORKUT, $c = 12$	1578 MB	1835 MB	2974 MB	2592 MB	3584 MB
ADDRESS, $c = 4$	202 MB	4400 MB	364 MB	336 MB	534 MB

DBLP	$c = 4$	$c = 6$	$c = 8$	$c = 10$	$c = 12$
SizeAware	2712 MB	1448 MB	579 MB	393 MB	320 MB

Table 4: The Memory Usage

The Scalability of R-S Join. We report the scalability of our SizeAware method for $\mathcal{R}$-$\mathcal{S}$ join in this section. We still varied the sizes of the datasets from 1 million to 3 millions for DBLP, ORKUT, and ADDRESS and from 250 thousand to almost 1 million for CLICK. We equally and randomly divided all the datasets into two parts for the $\mathcal{R}$-$\mathcal{S}$ join case and reported the elapsed time. The results are shown in Figure 12. We can see that the scalability of our method on $\mathcal{R}$-$\mathcal{S}$ join case was fairly good. For example, for DBLP dataset, when the threshold $c = 4$, the elapsed time for 1 million, 1.5 million, 2 million, 2.5 million, and 3 million sets were respectively 102 seconds, 180 seconds, 278 seconds, 405 seconds, and 518 seconds, which is consistent with our time complexity analysis. This is because the effectiveness of our proposed heuristics. In addition, the size boundary selection method can choose a good boundary for the SizeAware algorithm.

Comparing with Jaccard Set Similarity Join Methods. We compared SizeAware with six existing methods for set similarity join under Jaccard similarity constraint on the special ADDRESS dataset, which are AdaptJoin [30], GroupJoin [4], PPJoin [33], PPJoin+ [33], PEL [17], SKJ [31] (see Section 7 for the details of these methods). Figure 13(a) gives the results. Note PPJoin and PPJoin+ ran out of memory when $c = 2$ (which is the same as the threshold $\delta = 0.16$ for Jaccard similarity). We can see that SizeAware consistently outperformed the existing methods in all cases by up to 1-2 orders of magnitude. For example, when $c = 3$ (that is $\delta = 0.27$), the elapsed time for AdaptJoin, SKJ, PPJoin, PPJoin+, GroupJoin, PEL, and SizeAware was 1898s, 1318s, 2840s, 3024s, 1412s, 1608s, and 59s respectively. The reason was two-fold. First, when all the set sizes are exactly the same, the most effective length filter in all the existing work does not work. Second, all the existing methods use a filter-and-refine framework, which first generates candidates by some filtering conditions and then verifies the survived pairs. However, on ADDRESS dataset, the corresponding thresholds of the Jaccard similarity are low which limits the pruning power of the filtering conditions and makes the existing methods performed rather bad. Our SizeAware algorithm, however, directly generated all the result pairs *without generating candidates*. In other words, our method *does not require a verification step*, which is one of the reasons behind its efficiency. We also implemented the extensions for SizeAware as discussed in Appendix D to support Jaccard similarity and compared it with the six methods above on the other three datasets which have different set sizes. We varied the Jaccard similarity threshold from 0.8 to 0.95 and reported the elapsing time. Figure 13(b)-13(d) shows the results. Our SizeAware algorithm consistently outperformed the other methods. This is attributed to our proposed SizeAware algorithm and heap-based methods for reducing the filtering time and the cost-model in AdaptJoin for choosing the right value of c.