freshJoin: An Adaptive Algorithm for Set Containment Join

Jizhou Luo [*], Hong Gao, Jianzhong Li, Zening Zhou [†], Tao Zhang [‡]
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
{luojizhou, honggao, lijzh}@hit.edu.cn, {1666274344, zorseti}@qq.com

ABSTRACT

This paper revisits set containment join (SCJ) problem, which uses the subset relationship (*i.e.,* $\subseteq$) as condition to join set-valued attributes of two relations and has many fundamental applications in commercial and scientific fields. Existing in-memory algorithms for SCJ are either signature-based or prefix-tree-based. The former incurs high CPU costs because of hash code enumeration, while the latter incurs high space costs because of the storage of prefix trees. This paper proposes a new adaptive parameter-free in-memory algorithm, named as **fre**quency-ha**sh join** or FreshJoin in short, to evaluate SCJ efficiently. FreshJoin uses two flat indices to record three kinds of signatures(*i.e.,* the two least frequent elements and hash codes whose length is determined adaptively by the frequency of elements in the universe set). The first index is a sparse inverted index, which records which sets contain which elements or least frequent elements. And, the second index uses an array to record hash codes of all sets for each relation. Each item in the first index has a pointer pointing to an item in the second index. This pointer is exploited by FreshJoin to avoid enumerating of hash code. The time and space complexity of the proposed algorithm, which provide a rule to choose FreshJoin from existing algorithms, are analyzed. Experiments on 16 real-life datasets show that (1)FreshJoin usually reduces more than 70% of space costs and remains as competitive as the state-of-the-art algorithms in running time; (2) it remains as competitive as existing algorithms in both space costs and running time in the worst case.

PVLDB Reference Format:
Ben Trovato, G. K. M. Tobin, Lars Thørväld, Lawrence P. Leipuner, Sean Fogarty, Charles Palmer, John Smith, Julius P. Kumquat, and Ahmet Sacan. A Sample Proceedings of the VLDB Endowment Paper in LaTeX Format. *PVLDB*, 11(2): xxxx-yyyy, 2017.
DOI: https://doi.org/TBD

[*]Jizhou Luo is the corresponding author.
[†]Zening Zhou implements and debugs most methods.
[‡]Tao Zhang collects data and conducts the experiments.

Proceedings of the VLDB Endowment, Vol. 11, No. 2
Copyright 2017 VLDB Endowment 2150-8097/17/10... $ 10.00.
DOI: https://doi.org/TBD

1. INTRODUCTION

Sets are ubiquitous in computer science and set-valued attributes are widely used in database management systems, where data from commercial applications or scientific studies are processed and analyzed. For example, a set-valued attribute of a tuple may record the prerequisites of a course, or the labels imposed on a digital image, or the tokens in an email, or the outlinks of a webpage, and so on. The wide usage of set-valued attributes results in a large body of research interests on efficient algorithms for various kinds of fundamental operations on such attributes such as containment queries [1, 6, 8, 11, 19, 20, 25, 27], similarity joins [2, 3, 5, 21, 22, 23, 24], equality joins and containment joins [4, 7, 9, 10, 14, 15, 16, 17, 18, 26], which benefit many data processing and analytic tasks.

This paper focuses on set containment join (SCJ). That is, given two relations $\mathcal{R}$ and $\mathcal{S}$ with a set-valued attribute *set* each, to find all pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that $r.set \subseteq s.set$. For instance, in the online course selection system, each course has a set of prerequisites and each student has a set of courses he/she has learned. Let e_i denote a course. Fig. 1(a) illustrates prerequisites of courses in $\mathcal{R}$ and Fig. 1(b) shows learnt courses for each student in $\mathcal{S}$. Naturally, a student s can choose a course r only if s has studied all prerequisites of r(*i.e., r.set* $\subseteq$ *s.set*). By executing SCJ $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, the system can forecast all potential course selections and make arrangement for each course correspondingly.

Due to its fundamental nature, many efficient SCJ algorithms have been proposed [4, 7, 9, 10, 14, 15, 16, 17, 18, 26]. Among these, the early works are mainly disk-based algorithms. These algorithms are quite effective for joining massive set collections, and their performances are mainly bounded by their underlying in-memory processing strategies [14]. Besides, thanks to the improvement of modern hardware and the development of distributed computing infrastructures, recent work turn to study in-memory algorithms [4, 9, 10, 14, 26]. These work are either signature-based or prefix-tree-based.

Signature-based algorithms (e.g.,SHJ [7], PSJ [18], APSJ [17] and PTSJ [14]) encode the set of each tuple into a fixed-length signature(*i.e.,* bitmap), and use bitwise operation on the signatures as a filter to check possible containment, then verify the containment for the remaining tuple pairs. A common algorithmic challenge of these methods is how to find potential signatures pairs that may pass through the bitwise filter. The usual way is to, for each signature from $\mathcal{R}$, enumerate all potential signatures from $\mathcal{S}$, which incurs high CPU costs and works only on short signatures although special structures such as PATRICIA TRIE [14] can be used to alleviate this defect in some extent.

Most prefix-tree-based algorithms (*e.g.,*Pretti [9], Pretti+ [14], LIMIT [4], Piejoin [10]) use the ideas of *information retrieval* to evaluate SCJ. They usually build an inverted index I on $\mathcal{S}$, where $I_{\mathcal{S}}(e_i)$ records all $s \in \mathcal{S}$ with $e_i \in s.set$. Then, for each $r \in \mathcal{R}$, they compute the intersection $\cap_{e_i \in r} I_{\mathcal{S}}(e_i)$ of inverted lists related to r and output $\langle r, s\rangle$ for each s in the intersection. The prefix tree is exploited to accelerate the speed of algorithms by sharing the intersection operations among any tuples $r \in \mathcal{R}$ which have common prefixes. For example, when each tuple of the sample relation $\mathcal{R}$ in Fig. 1(a) is visited by traversing the prefix tree in Fig. 1(c) depth-firstly, the intersection of inverted lists $I_{\mathcal{S}}(e_i)$ are computed correspondingly. And, the intersection $I_{\mathcal{S}}(e_1) \cap I_{\mathcal{S}}(e_3)$ at the grey-colored node is shared by its descendants. The state-of-the-art prefix-tree-based algorithm ttjoin [26] takes k-least frequent elements of sets in $\mathcal{R}$ as their signatures and indexes signatures in a prefix tree $T_{\mathcal{R}}$. Besides, all sets in $\mathcal{S}$ are also indexed in a prefix tree $T_{\mathcal{S}}$. Then, ttjoin traverses $T_{\mathcal{S}}$ depth-firstly to visit each set $s.set$ in $\mathcal{S}$ and traverses $T_{\mathcal{R}}$ in the same way to locate all r in $\mathcal{R}$ such that $r.set \subseteq s.set$.

All these algorithms perform SCJ efficiently with high space costs because of the storage of prefix tree. In fact, when the average set size is large, the space for prefix tree may reach several times of space for the data itself. Although the space costs can be sharply reduced by limiting the height of the tree (*e.g.,*LIMIT [4], ttjoin [26]), or storing the tree as several arrays (*e.g.,*Piejoin [10]), or compressing the non-branching nodes into single nodes (Pretti+ [14]), these algorithms do not perform well on all datasets because their performance depend on some empirical parameters (*e.g.,* the height of prefix trees) or other factors which are hardly adaptive to datasets themselves.

In big data era, it is important to make SCJ-algorithms well-scaled in both space costs and running time. To do so, this paper proposes a new parameter-free adaptive algorithm, named as **fre**quency-ha**sh join** or FreshJoin in short, to evaluate SCJ efficiently. FreshJoin gives up prefix trees totally to reduce space costs. Instead, it uses two flat indices to record three kinds of signatures. The first type of signatures is the hash signatures(*i.e.,* bitmaps) of sets in both $\mathcal{R}$ and $\mathcal{S}$, the second (third *resp.*) type of signatures is the (2nd *resp.*) least frequent elements of sets in $\mathcal{R}$. These signatures are well organized in the new index structure such that (1) FreshJoin can use the bitwise filter (like in SHJ [7] and PTSJ [14]) but without enumerating the hash codes; and (2) FreshJoin can exploit the hash signatures to reduce as many as possible tuple pairs fed into the bitwise filter. This guarantees that FreshJoin evaluates SCJ efficiently.

Moreover, FreshJoin performs SCJ adaptively according to the statistics of the datasets by allowing the lengths of hash codes to change adaptively. The statistics used in FreshJoin mainly include the average and standard deviation of the sizes of sets, the average and standard deviation of the frequencies of the elements in the universe set, and so on. FreshJoin uses these information to differ low-frequency elements and high-frequency elements from mid-frequency elements, and hashes them separately. The signature length of each part can also be estimated adaptively by a new defined hash function, which makes the total signature length far smaller than that estimated in PTSJ [14].

FreshJoin always keeps high efficiency adaptively. Compare with the state-of-the-art SCJ algorithms, FreshJoin usually keeps as competitive as its counterparts in running time and reduces even more than 70% of space costs. In the worst case, it remains as competitive as its counterparts in both space costs and running time. Our theoretical analysis provides a rule to distinguish the worst case from other cases.

The principle contributions of this paper are summarized as follows.

- We propose a new parameter-free adaptive algorithm to support set containment join efficiently with low space costs. We state the correctness of the algorithm, and analyze its complexity which provides a rule to choose our algorithm from existing algorithms.
- we propose a sparse asymmetric inverted index to make three kinds of signatures work coordinately and space economically. We analyze the space costs of the index.
- We propose a new hash function to estimate the signature length adaptively according to the statistics of the datasets.
- We have implemented our method. Experimental results on 16 real-life representative datasets show that our parameter-free SCJ algorithm is adaptive, well scaled, efficient and effective.

The rest of this paper is organized as follows. Section 2 is preliminaries. Section 3 describes the framework of FreshJoin algorithm. The hash function and the signature length are discussed in Section 4. Section 5 reports experimental results. Section 6 summaries the related work, followed by the conclusion in Section 7.

2. PRELIMINARIES

In this section, we introduce basic concepts and definitions used in the paper. Table 1 summarizes the important mathematical notations used throughout this paper.

We assume a discrete universe set, denoted as $\mathcal{U}$, consisting of a linearly ordered list of elements $e_1, e_2, \cdots, e_n$. n is called as the size of $\mathcal{U}$, denoted as $n = |\mathcal{U}|$. A subsequence $e_{k_1}, e_{k_2}, \cdots, e_{k_m}$ of $e_1, e_2, \cdots, e_n$ with $1 \le k_1 < k_2 < \cdots < k_m \le n$ is called as a set from universe $\mathcal{U}$ (or a set in short), and m is called as the size of the set. The set of size 0 is empty set. The collection of all sets from $\mathcal{U}$ is denoted as $2^{\mathcal{U}}$. A set can be denoted by single characters such as $A, B, \cdots$, *i.e.,* $A = \{e_{k_1}, e_{k_2}, \cdots, e_{k_m}\}$ means A is the set $e_{k_1}, e_{k_2}, \cdots, e_{k_m}$. $e_i \in A$ means elements e_i appears in set A and $e_i \notin A$ means otherwise. The i-th element in A is denoted as $e_i^{(A)}$, *i.e.,* $e_i^{(A)}{=}e_{k_i}$. The size of a set A is often denoted as $|A|$. If each element appearing in set A also appears in set B, we call A is a subset of B, denoted as $A \subseteq B$, which can be verified in $O(|A| + |B|)$ time as follows.

```
Procedure verify $(A, B)$
Input: two sets $A$ and $B$
output: whether $A \subseteq B$ or not
1.$j \leftarrow 1$;
2.For $i \leftarrow 1$ To $|A|$
3.    While $j \le |B|$ and $e_i^{(A)} \ne e_j^{(B)}$ Do $j$++;
4.    If $j > |B|$ then return false;
5.return true
```

Set-valued relations associate each tuple with a set from $\mathcal{U}$. The schemas of set-valued relations are represented as

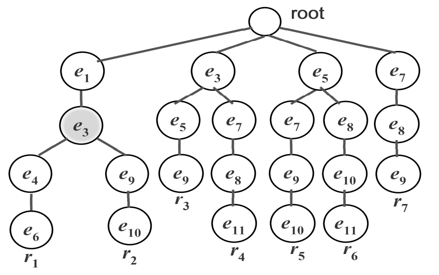
id	set
r_1	$\{e_1,e_3,e_4,e_6\}$
r_2	$\{e_1,e_3,e_9,e_{10}\}$
r_3	$\{e_3,e_5,e_9\}$
r_4	$\{e_3,e_7,e_8,e_{11}\}$
r_5	$\{e_5,e_7,e_9,e_{10}\}$
r_6	$\{e_5,e_8,e_{10},e_{11}\}$
r_7	$\{e_7,e_8,e_9\}$

(a) A sample relation $\mathcal{R}$

id	set	id	set
s_1	$\{e_1,e_3,e_5,e_6,e_9,e_{11}\}$	s_7	$\{e_4,e_6,e_7,e_{10},e_{11}\}$
s_2	$\{e_2,e_4,e_5,e_9,e_{10},e_{11}\}$	s_8	$\{e_4,e_7,e_8,e_{10},e_{11}\}$
s_3	$\{e_2,e_5,e_7,e_9,e_{10},e_{11}\}$	s_9	$\{e_5,e_6,e_8,e_9,e_{10},e_{11}\}$
s_4	$\{e_3,e_7,e_8,e_9,e_{10},e_{11}\}$	s_{10}	$\{e_6,e_7,e_8,e_{10},e_{11}\}$
s_5	$\{e_3,e_8,e_9,e_{10},e_{11}\}$	s_{11}	$\{e_6,e_8,e_9,e_{10},e_{11}\}$
s_6	$\{e_4,e_5,e_6,e_7,e_8,e_9\}$	s_{12}	$\{e_7,e_8,e_9,e_{10},e_{11}\}$

The fequency of e_i equals i for i=1,2,...,11

(b) A sample relation $\mathcal{S}$



(c) The prefix tree of $\mathcal{R}$

Figure 1: Two simple relations and a prefix tree

$\mathbb{R} = (\mathbb{A}_1, \cdots, \mathbb{A}_p, SET)$, where $\mathbb{A}_i$ is an attribute with domain Ω_i for $i = 1, \cdots, p$ and SET is an attribute with domain $2^{\mathcal{U}}$. A tuple r over schema $\mathbb{R}$ is a finite collection that contains for each $\mathbb{A}_i$ a value $v_i \in \Omega_i$ and for SET an set $r.set$ from the universe $\mathcal{U}$. The i-th element of $r.set$ is denoted as $e_i^{(r)}$ without any ambiguity. A set-valued relation $\mathcal{R}$ over schema $\mathbb{R}$ is a finite collection of tuples over $\mathbb{R}$. The size of $\mathcal{R}$, denoted as $|\mathcal{R}|$, is the number of tuples in $\mathcal{R}$.

Definition 1: (**Set Containment Join**) Given two set-valued relations $\mathcal{R}$ and $\mathcal{S}$, the set containment join (or SCJ in short) between $\mathcal{R}$ and $\mathcal{S}$, denoted as $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, is to find all tuple-pairs $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$ such that $r.set \subseteq s.set$. That is $\mathcal{R} \bowtie_{\subseteq} \mathcal{S} = \{\langle r, s\rangle | r \in \mathcal{R}, s \in \mathcal{S}, r.set \subseteq s.set\}$. □

Example 1: For relations $\mathcal{R}$ (Fig. 1(a)) and $\mathcal{S}$ (Fig. 1(b)), the result of set containment join $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$ is $\{\langle r_3, s_1\rangle, \langle r_4, s_4\rangle, \langle r_5, s_3\rangle, \langle r_6, s_9\rangle, \langle r_7, s_4\rangle, \langle r_7, s_6\rangle, \langle r_7, s_{12}\rangle\}$. □

In set containment join, we assume the input relations $\mathcal{R}$ and $\mathcal{S}$ share a common universe set $\mathcal{U}$. In fact, any tuples $r \in \mathcal{R}$, whose set $r.set$ contains elements not in the universe of $\mathcal{S}$, can be removed from $\mathcal{R}$ by data loading procedure without affecting the join results. Moreover, we assume all elements in the shared universe $\mathcal{U}$ are sorted in ascending order of their frequencies, which is defined below.

Definition 2: (**frequency of element**) Given the input relations $\mathcal{R}$ and $\mathcal{S}$ of SCJ $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, the $\mathcal{S}$-frequency (or frequency in short) of each element e_i in the universe $\mathcal{U}$, denoted as $f_{\mathcal{S}}(e_i)$, is the number of tuples $s \in \mathcal{S}$ such that $e_i \in s.set$. That is $f_{\mathcal{S}}(e_i) = |\{s | s \in \mathcal{S}, e_i \in s.set\}|$. □

Example 2: For relations $\mathcal{R}$ (Fig. 1(a)) and $\mathcal{S}$ (Fig. 1(b)), element e_i in the shared universe $\mathcal{U} = \{e_1, e_2, \cdots, e_{11}\}$ has $\mathcal{S}$-frequencies i for $i = 1, 2, ..., 11$. And, $\mathcal{U}$ is sorted in increasing order of elements' frequencies. □

Based on the sorted universe, tuples in both input relations of SCJ can be sorted further by the data loading procedure in *lexicographical order* of their sets. Most SCJ algorithms (*e.g.,*Pretti [9], Pretti+ [14], LIMIT [4],Piejoin [10] and ttjoin [26]) benefit from this by accelerating both the creation of prefix tree and the joining procedure. This paper also assume the input relations be sorted in this way. For instance, $\mathcal{R}$ (Fig. 1(a)) and $\mathcal{S}$ (Fig. 1(b)) are sorted.

A naive SCJ algorithm applies procedure verify on each pair $\langle r, s\rangle \in \mathcal{R} \times \mathcal{S}$, and results in $O(|\mathcal{R}| \cdot |\mathcal{S}| \cdot l_{avg}(\mathcal{S}))$ time. Hash signatures can be used to accelerate this algorithm by adding a bitwise filter before applying verify.

Hash signatures of sets are bitmaps of length $w_{sig} \cdot 64$. They are used to represent or approximate sets in w_{sig} 64-bit integers. For set containment join, it suffices if we set one bit in the hash signature for each element of the set whose signature we want to compute. A function *hash* is applied to map each element to an integer in interval $[0, w_{sig} * 64 - 1]$. Thus, the hash signature $sig(set)$ of a set set can be computed by successively setting $hash(e_i)$-th bit for each element $e_i \in set$. Such hash signatures are all SCJ-friendly, *i.e.,* $set_1 \subseteq set_2 \Rightarrow sig(set_1) \& sig(set_2) = sig(set_1)$, where & is the bitwise AND operation.

Example 3: Here is a toy hash function h, which generates 8-bit SCJ-friendly hash signatures for data in Fig. 1. $h(e_1)$=0. $h(e_2)$=$h(e_3)$=1. $h(e_4)$=$h(e_5)$=2. $h(e_6)$=3. $h(e_7)$ =$h(e_8)$=4. $h(e_9)$=5. $h(e_{10})$=6 and $h(e_{11})$=7. Under this function, since r_6=$\{e_5, e_8, e_{10}, e_{11}\}$, only the 3rd, 5th, 7th and 8th bit in its hash signature are 1, *i.e.,* $sig(r_6)$ =00101011. Similarly, $sig(s_9)$=00111111 and $sig(s_{10})$=00011011. Since $sig(r_6)\&sig(s_9)$=$sig(r_6)$, $\langle r_6, s_9\rangle$ may belong to $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$. While $sig(r_6)\&sig(s_{10}) \neq sig(r_6)$, $\langle r_6, s_{10}\rangle$ must not belong to $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$. □

SCJ-friendly hash signatures can be used to accelerate set containment join algorithms, because most tuple-pairs $\langle r, s\rangle$ with $r.set \nsubseteq s.set$ can be pruned away in $O(w_{sig})$ time via the bitwise filter below.

```
Procedure bitwiseFilter ($sig_1, sig_2$)
Input: signatures $sig_1$ and $sig_2$ of length $w_{sig} \times 64$
output: whether $sig_1 \& sig_2 = sig_1$ or not
1.For $i \leftarrow 0$ To $w_{sig} - 1$
2.    If $sig_1[i] \& sig_2[i] \neq sig_1[i]$ then return false;
3.return true
```

With the help of hash signatures, SCJ can be evaluated by the filter-and-refine framework below, where Line 3 enumerates all $s \in \mathcal{S}$ probably satisfying $sig(r)\&sig(s) = sig(r)$. Unlike the approaches adopted in SHJ [7] and PTSJ [14], this paper uses smart mechanisms to avoid such enumerations by establishing connections from hash signatures of $r \in \mathcal{R}$ to hash signatures of such $s \in \mathcal{S}$.

```
Filter-And-Refine Framework: SCJ ($\mathcal{R}$,$\mathcal{S}$)
Input: two set-valued relations $\mathcal{R}$ and $\mathcal{S}$
output: $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
1.obtain hash signature for each $r \in \mathcal{R}$,$s \in \mathcal{S}$;
2.For each $r \in \mathcal{R}$ Do
3.    For $s \in \mathcal{S}$ with $sig(r)\&sig(s) = sig(r)$ Do
4.      If verify($r$,$s$) then output $\langle r, s\rangle$;
```

3. FRAMEWORK OF FRESHJOIN

In this section, we present the framework of the FreshJoin algorithm. We first describe the index structure, discuss how to build this index and analyze its space costs (Section 3.1). Then, we present FreshJoin algorithm, state its correct-

Table 1: The summary of Notations

Notation	Definition
$\mathcal{U}$; $\|\mathcal{U}\|$	universe set; size of universe set
r, $\mathcal{R}$; s, $\mathcal{S}$	a tuple, a set-valued relation
$\|\mathcal{R}\|$; $\|\mathcal{S}\|$	number of tuples in a set-valued relation
$f_{\mathcal{S}}(e_i)$	frequency of element e_i in $\mathcal{S}$
$I_{\mathcal{R}}(e_i)$	inverted list of element e_i for tuples in $\mathcal{R}$
$I_{\mathcal{S}}(e_i)$	inverted list of element e_i for tuples in $\mathcal{S}$
$sig_{\mathcal{R}}(r_j)$	hash code of a set $r_j.set$ for $r_j \in \mathcal{R}$
$sig_{\mathcal{S}}(s_j)$	hash code of a set $s_j.set$ for $s_j \in \mathcal{S}$
$e_j^{(r)}$; $e_j^{(s)}$	j-th element in $r.set$, $s.set$ for $r \in \mathcal{R}$,$s \in \mathcal{S}$
$l_{avg}(\mathcal{S})$	average size of sets $s.set$ for $s \in \mathcal{S}$
$\sigma_{\mathcal{S}}$	standard deviation of all $\|s.set\|$ for $s \in \mathcal{S}$
M	$e_M \in \mathcal{U}$ is the first mid frequency element
H	$e_H \in \mathcal{U}$ is the first high frequency element
w_{sig}	length of hash code, in unit of 64-bit integer
M'	$b_0 \sim b_{M'-1}$ in signature is for low frequency
H'	$b_{M'} \sim b_{H'-1}$ in signature is for low frequency

ness and analyze its complexity (Section 3.2). The detail of hash function is postponed to next section, except that the signature length w_{sig} (in unit of 64-bit integers) is assumed.

3.1 The Index and Its Creation

Three kinds of signatures play different roles in FreshJoin algorithm. The first is the hash codes, each denoted as w_{sig} 64-bit integers, associated with tuples in both $\mathcal{R}$ and $\mathcal{S}$. Such signatures will be fed into bitwise filters of FreshJoin to prune away most tuple-pairs that are not join results. The second is the least frequent element in set $r.set$ for each $r \in \mathcal{R}$. Such signatures associate the first type of signatures for tuples in $\mathcal{R}$ with those for tuples in $\mathcal{S}$, and provide opportunity to use hash code in set containment join without enumerating them. The third is the 2nd least frequent elements in set $r.set$ for each $r \in \mathcal{R}$. Such signatures are used to accelerate the join procedure by reducing hash code fed into the bitwise filter. To make these signatures work coordinately, they are well organized into two kinds of flat index structures, *i.e.,* arrays and inverted indices.

The Structure of freshIndex. Two arrays $sig_{\mathcal{R}}$ and $sig_{\mathcal{S}}$ are used to index the first kind of signatures for $\mathcal{R}$ and $\mathcal{S}$ respectively. Each unit of the arrays is w_{sig} 64-bit integers, and can be accessed by using the IDs of tuples in $\mathcal{R}$ or $\mathcal{S}$.

A sparse inverted index $I_{\mathcal{R}}$ is used to index the 2nd and 3rd kind of signatures for tuples in $\mathcal{R}$. Each element $e \in \mathcal{U}$ has an inverted list $I_{\mathcal{R}}(e)$ in $I_{\mathcal{R}}$. Each item in $I_{\mathcal{R}}(e)$ is a pair $\langle i, e' \rangle$, which means that $r_i \in \mathcal{R}$ and e (e' *reps.*) is the (2nd *resp.*) least frequent element in $r_i.set$. All items in each list is sorted such that items with a same second component are stored contiguously. In this way, all tuples r with $r.set$ having the same 2nd least frequent element can be processed by FreshJoin in a batched manner. Notice that, each $r \in \mathcal{R}$ is indexed only once in $I_{\mathcal{R}}$. Since sets may share a common least frequent element, many $I_{\mathcal{R}}(e)$s may be null. This is why $I_{\mathcal{R}}$ is sparse. Strictly speaking, $I_{\mathcal{R}}$ is not an inverted index. We still call it like this, just because it works like an inverted index.

Besides, an other sparse inverted index $I_{\mathcal{S}}$ is used to index the tuples in $\mathcal{S}$. Each element $e \in \mathcal{U}$ has an inverted list $I_{\mathcal{S}}(e)$ in $I_{\mathcal{S}}$. Each item in $I_{\mathcal{S}}(e)$ is a tuple ID i, which means that $s_i \in \mathcal{S}$ and $e \in s_i.set$. Each list $I_{\mathcal{S}}(e)$ is sorted

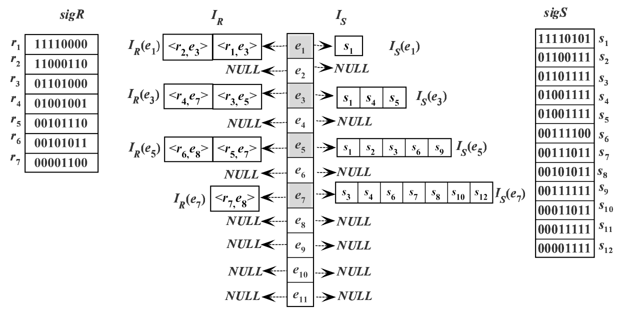


Figure 2: freshIndex of sample relations in Fig. 1

in an ascending order of its items. In this way, the time cost of computing the intersection of two inverted lists is proportional to the sum of their lengths. $I_{\mathcal{S}}$ is also sparse, because we do not create some inverted lists $I_{\mathcal{S}}(e)$ at all if they are useless in FreshJoin algorithm. Specifically, $I_{\mathcal{S}}(e)$ is null if $e \in \mathcal{U}$ is not the 2nd or 3rd type of signatures for any tuple $r \in \mathcal{R}$, *i.e.,* e is neither of the two least frequent elements in $r.set$ for any $r \in \mathcal{R}$.

Example 4: Fig. 2 illustrates the index structure with data in Fig. 1. Array $sig_{\mathcal{R}}$ ($sig_{\mathcal{S}}$ *resp.*) stores hash signatures of tuples in $\mathcal{R}$ ($\mathcal{S}$ *resp.*). These signatures are generated via the toy hash function in Example 3. In the sparse inverted index $I_{\mathcal{R}}$, $I_{\mathcal{R}}(e_1)$ contains two items $\langle r_1, e_3 \rangle$ and $\langle r_2, e_3 \rangle$, since only sets of $r_1, r_2 \in \mathcal{R}$ have e_1 as their least frequent elements. $I_{\mathcal{R}}(e_2)$ is null, since $\mathcal{R}$ has no tuple with e_2 as its least frequent elements. Other lists in $I_{\mathcal{R}}$ is similar. While in the sparse inverted index $I_{\mathcal{S}}$, $I_{\mathcal{S}}(e)$ is non-null only if $I_{\mathcal{R}}(e)$ is. For instance, $I_{\mathcal{S}}(e_3)$ contains three items, because $I_{\mathcal{R}}(e_3)$ is not null and only the sets of $s_1, s_4, s_5 \in \mathcal{S}$ contains e_3. Although e_{11} is the most frequent element, $I_{\mathcal{S}}(e_{11})$ is null because $I_{\mathcal{R}}(e_{11})$ is null. Thus, we have an intuition that sparsity of $I_{\mathcal{R}}$ and $I_{\mathcal{S}}$ helps to save a lot of spaces. □

The Creation of freshIndex. The idea to create freshIndex is rather simple. Both relation $\mathcal{R}$ and $\mathcal{S}$ are sorted in lexicographical order of their sets first, which guarantees that inverted lists in both $I_{\mathcal{R}}$ and $I_{\mathcal{S}}$ are ordered naturally. Then, $\mathcal{R}$ is indexed and elements in the universe are marked at the same time. Finally, the $\mathcal{S}$ is indexed according to the marked elements.

Alg. 1 implements the ideas above. It indexes $\mathcal{R}$ in Line 1-Line 9 and indexes $\mathcal{S}$ in Line 10-Line 15. When $\mathcal{R}$ is indexed, it is sorted lexicographically first (Line 1), and then each tuple $r_i \in \mathcal{R}$ is indexed (Line 2-Line 9). For r_i, the hash code of $r_i.set$ is obtained by invoking hashAset (see Section 4) and stored in the array unit $sig_{\mathcal{R}}(r_i)$ (Line 3). After that, r_i is indexed in $I_{\mathcal{R}}$ (Line 4-Line 9). If $|r_i.set| = 1$(*i.e.,* it contains only one element), then item $\langle r_i.ID, - \rangle$ is appended to the end of $I_{\mathcal{R}}(e_1^{(r_i)})$ (Line 5) and $e_1^{(r_i)}$ is marked (Line 6). Otherwise, item $\langle r_i.ID, e_2^{(r_i)} \rangle$ is appended to the end of $I_{\mathcal{R}}(e_1^{(r_i)})$ (Line 8), and both $e_1^{(r_i)}$ and $e_2^{(r_i)}$ are marked (Line 9). When $\mathcal{S}$ is indexed, it is also sorted lexicographically first(Line 10), and then each tuple $s_i \in \mathcal{S}$ is indexed (Line 11-Line 15). The hash code of $s_i.set$ is obtained in the same way but stored in $sig_{\mathcal{S}}(s_i)$ (Line 12). After that, the ID of s_i is added into some inverted lists according to the elements in $s_i.set$ are marked or not (Line 14-Line 15). Specifically, if the j-th element $e_j^{(s_i)}$ of $s_i.set$ is marked, then i is added to the tail of $I_{\mathcal{S}}(e_j^{(s_i)})$.

Example 5: To build freshIndex for data in Fig. 1, the hash signatures of tuples in $\mathcal{R}$ and $\mathcal{S}$ are generated as illustrated in Example 3, and stored into arrays $sig_\mathcal{R}$ and $sig_\mathcal{S}$ respectively (see Fig. 2). After that, each $r_i.set$ ($r_i \in \mathcal{R}$) is indexed into $I_\mathcal{R}$ one by one in Line 2-Line 9 of Alg. 1, elements e_1, e_3, e_5 and e_7 are marked in order. Thus, when $s_i.set$ ($s_i \in \mathcal{R}$) is indexed into $I_\mathcal{S}$ in Line 10-Line 15, only lists of these marked elements are filled. For example, although s_1 contains 6 elements, $s_1.ID$ only appears in $I_\mathcal{S}(e_1)$, $I_\mathcal{S}(e_3)$ and $I_\mathcal{S}(e_5)$. □

Algorithm 1: freshIndex ($\mathcal{R}$,$\mathcal{S}$)

```
    Input: two set-valued relations $\mathcal{R}$ and $\mathcal{S}$
    Output: fresh index of set containment join $\mathcal{R} \bowtie_\subseteq \mathcal{S}$
1   sort all tuples in $\mathcal{R}$ lexicographically as $r_1, \cdots, r_{|\mathcal{R}|}$;
2   for $i \leftarrow 1$ To $|\mathcal{R}|$ do
3       $sig_\mathcal{R}(r_i) \leftarrow$ hashAset($r_i.set$);          // see sec.4
4       if $|r_i.set| = 1$ then
5           add $\langle r_i.ID, -\rangle$ to the end of $I_\mathcal{R}(e_1^{(r_i)})$;
6           mark $e_1^{(r_i)}$;
7       else
8           add $\langle r_i.ID, e_2^{(r_i)}\rangle$ to the end of $I_\mathcal{R}(e_1^{(r_i)})$;
9           mark both $e_1^{(r_i)}$ and $e_2^{(r_i)}$;
10  sort all tuples in $\mathcal{S}$ lexicographically as $s_1, \cdots, s_{|\mathcal{S}|}$;
11  for $i \leftarrow 1$ To $|\mathcal{S}|$ do
12      $sig_\mathcal{S}(s_i) \leftarrow$ hashAset($s_i.set$);          // see sec.4
13      for $j \leftarrow 1$ To $|s_i.set|$ do
14          if $e_j^{(s_i)}$ is marked then
15              add $s_i.ID$ to the end of $I_\mathcal{S}(e_j^{(s_i)})$;
```

Analysis. It is straightforward to verify that the output of Alg. 1 is the expected index structure. The time complexity of Alg. 1 is postponed to Section 4 till the procedure hashAset is clear. Now, we answer questions below. Can the design of the sparse inverted indices really save any space? And, how much space do they save?

In a usual inverted index, each inverted list $I_\mathcal{S}(e)$ is not null if there is at least one tuple $s \in \mathcal{S}$ such that $e \in s.set$ for any $e \in \mathcal{U}$. However, in the freshIndex, $I_\mathcal{S}(e)$ is null if there is no tuple $r \in \mathcal{R}$ such that $r.set$ contains e as one of its two least frequent elements, which is the often case for these highest frequent elements. The space costs of $I_\mathcal{S}$ is determined by the number of marked elements during indexing $\mathcal{R}$. To simplify the analysis, we assume each set $r.set$ ($r \in \mathcal{R}$) be chosen from $\mathcal{U}$ uniformly and independently, and obtain the lemma below, whose proof can be found in Appendix A.

Lemma 1: *Assume each set $r.set$ ($r \in \mathcal{R}$) be sampled from $\mathcal{U}$ uniformly and independently, then at most $0.9 \cdot |\mathcal{R}|$ elements in $\mathcal{U}$ are marked by Line 6 or Line 9 of Alg. 1.*

Since $|\mathcal{U}|$ is an other upper bound, the number of marked elements does not exceed $\min(|\mathcal{U}|, 0.9 \cdot |\mathcal{R}|)$. Moreover, since each set $s.set$ ($s \in \mathcal{S}$) contains $l_{avg}(S)$ elements on average, there are totally $|\mathcal{S}| \cdot l_{avg}(S)$ items in a usual inverted index of $\mathcal{S}$ and each inverted list has $\frac{|\mathcal{S}| \cdot l_{avg}(S)}{|\mathcal{U}|}$ items on average. Therefore, $I_\mathcal{S}$ has $\min(|\mathcal{U}|, 0.9 \cdot |\mathcal{R}|) \cdot \frac{|\mathcal{S}| \cdot l_{avg}(S)}{|\mathcal{U}|}$ items on average.

Besides, $I_\mathcal{R}$ needs $O(|\mathcal{R}|)$ space, because each r_i is indexed only once. $sig_\mathcal{R}$ and $sig_\mathcal{S}$ need $O(w_{sig} \cdot (|\mathcal{R}| + |\mathcal{S}|))$ space.

Put together, we obtain the theorem below.

Theorem 2: *The freshIndex of relations $\mathcal{R}$ and $\mathcal{S}$ needs $O((2+w_{sig}) \cdot |\mathcal{R}| + [\frac{\min(|\mathcal{U}|, 0.9|\mathcal{R}|)}{|\mathcal{U}|} \cdot l_{avg}(\mathcal{S}) + w_{sig}] \cdot |\mathcal{S}|)$ space.*

Remark. Notice that $\frac{\min(|U|, 0.9|\mathcal{R}|)}{|\mathcal{U}|} \leq 1$. Theorem 2 tells us that: (1) when $|\mathcal{R}| \leq |\mathcal{U}|$, freshIndex really saves space; (2) When $|\mathcal{R}| >> |\mathcal{U}|$, freshIndex is a usual inverted index and can not save space; (3) When both w_{sig} and $l_{avg}(S)$ are constants, freshIndex only needs linear space. These conclusions explain well the experimental results in Section 5.

3.2 The Join Algorithm

The basic idea of FreshJoin is similar to that of SHJ [7]. That is to use bitwise operations on hash signatures of sets to prune away as many as possible set-pairs whose subset relationship are verified, because bitwise filter is much more economic than the verification. FreshJoin accomplishes this efficiently by applying appropriately the three kinds of signatures indexed in freshIndex. In one hand, it uses the 2nd type of signatures of tuples in $\mathcal{R}$ (*i.e.,* the least frequent element in their sets) to locate the hash-code-pairs, which are fed into the bitwise filter, by only joining tuples indexed in $I_\mathcal{R}(e)$ and $I_\mathcal{S}(e)$ for the same es. This is feasible because any set $s.set$ ($s \in \mathcal{S}$) with $s.set \supseteq r.set$ ($r \in \mathcal{R}$) must contain the least frequent element in $r.set$. In the other hand, since $I_\mathcal{S}(e)$ may be very long and comparisons between tuple IDs are often more economic than bitwise filter, FreshJoin exploits the 3rd type of signatures of tuples in $\mathcal{R}$ to reduce the number of hash-signature-pairs by computing the intersection of $I_\mathcal{S}(e)$ and $I_\mathcal{S}(e')$, where e's are the second component of indexed items in $I_\mathcal{R}(e)$.

Alg. 2 implements the ideas above. First of all, it calls freshIndex to build index of the input relations (Line 1), and initializes the output set J (Line 2). Then, it processes each pair $\langle I_\mathcal{R}(e_i), I_\mathcal{S}(e_i)\rangle$ of inverted lists sequentially (Line 3-Line 14). Null inverted lists are skipped over (Line 4). Indexed items in each non-null list $I_\mathcal{R}(e_i)$ are processed one by one (Line 7- Line 14). For each item $\langle j, e_u\rangle \in I_\mathcal{R}(e_i)$, it determines whether $I_\mathcal{S}(e_i) \cap I_\mathcal{S}(e_u)$ needs to be computed, according to whether e_u has been encountered or not. If yes, it does not compute the intersection and skips over Line 8-Line 10. Otherwise, it computes the intersection (Line 9) and traces the new encountered 2nd least frequent element (Line 10). Next, for each remaining tuple ID $k \in List$, it accesses the arrays $\langle sig_\mathcal{R}, sig_\mathcal{S}\rangle$ to obtain a hash-signature-pair, and feeds this pair into the bitwise filter (Line 13). Finally, the surviving tuple-pair $\langle r_j, s_k\rangle$ are verified and added into J if $r_j.set \subseteq s_k.set$ (Line 14).

Example 6: Consider the join of sample relations in Fig. 1 with the help of freshIndex in Fig. 2. The inverted lists $I_\mathcal{R}(e_i)$ and $I_\mathcal{S}(e_i)$ are joined for each e_i. When e_1 is considered, since $\langle r_1, e_3\rangle \in I_\mathcal{R}(e_1)$, the intersection $I_\mathcal{S}(e_1) \cap I_\mathcal{S}(e_3) = List = \{s_1\}$ is computed. Since $sig_\mathcal{R}(r_1)\&\ sig_\mathcal{S}(s_1) \neq sig_\mathcal{R}(r_1)$, pair $\langle r_1, s_1\rangle$ is pruned away. For the second item $\langle r_2, e_3\rangle \in I_\mathcal{R}(e_1)$, since it contains the same element e_3 as the first item, the intersection is not recomputed. Thus, *List* is still $\{s_1\}$. Since $sig_\mathcal{R}(r_2)\&sig_\mathcal{S}(s_1) \neq sig_\mathcal{R}(r_2)$, pair $\langle r_2, s_1\rangle$ is also pruned away. After all lists are considered similarly, the join result shown in Example 1 can be obtained. □

Correctness. We assert that the output J of Alg. 2 is exactly $\mathcal{R} \bowtie_\subseteq \mathcal{S}$. It is obvious that $J \subseteq \mathcal{R} \bowtie_\subseteq \mathcal{S}$, because $r.set \subseteq s.set$ is verified in Line 14 for any $\langle r, s\rangle \in J$. Reversely,

Algorithm 2: freshjoin $(\mathcal{R},\mathcal{S})$

```
Input: two set-valued relations $\mathcal{R}$ and $\mathcal{S}$
Output: the result set $J$ of $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
1  $I_{\mathcal{R}}, sig_{\mathcal{R}}, I_{\mathcal{S}}, sig_{\mathcal{S}} \leftarrow$ freshIndex $(\mathcal{R},\mathcal{S})$;
2  $J \leftarrow \emptyset$;
3  for $i \leftarrow 1$ To $|\mathcal{U}|$ do
4      If $I_{\mathcal{R}}(e_i) = NULL$ then continue;
5      $e \leftarrow$-;
6      $List \leftarrow I_{\mathcal{S}}(e_i)$;
7      foreach $\langle j, e_u\rangle \in I_{\mathcal{R}}(e_i)$ do
8          if $e \neq e_u$ then
9              $List \leftarrow I_{\mathcal{S}}(e_i) \cap I_{\mathcal{S}}(e_u)$;
10             $e \leftarrow e_u$;
11         foreach $k \in List$ do
12             If $sig_{\mathcal{R}}(r_j) \& sig_{\mathcal{S}}(s_k) \neq sig_{\mathcal{S}}(r_j)$ then
13                 continue;
14             If $verify(r_j, s_k)$ then $J \leftarrow J \cup \{\langle r_j, s_k\rangle\}$;
15 return $J$;
```

if $\langle r,s\rangle \in \mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, We show $\langle r,s\rangle \in J$ as follows. Without loss of generality, assume $|r.set| \geq 2$. First of all, $\langle r.ID, e_2^{(r)}\rangle$ is indexed in list $I_{\mathcal{R}}(e_1^{(r)})$, according to Alg. 1. Of course, both $e_1^{(r)}$ and $e_2^{(r)}$ are marked. Now that $r.set \subseteq s.set$, we have $e_1^{(r)} \in s.set$ and $e_2^{(r)} \in s.set$. Moreover, $s.ID$ is indexed in both $I_{\mathcal{S}}(e_1^{(r)})$ and $I_{\mathcal{S}}(e_2^{(r)})$, according to Line 13-Line 15 of Alg. 1. Therefore, $s.ID$ appears in $List = I_{\mathcal{S}}(e_1^{(r)}) \cap I_{\mathcal{S}}(e_2^{(r)})$ when item $\langle r.ID, e_2^{(r)}\rangle$ is processed in Line 7-Line14 of Alg. 2. Since $r.set \subseteq s.set$ and the signature is SCJ-friendly, $\langle r,s\rangle$ passes through the bitwise filter (Line 12) and containment verification (Line 14). Thus, $\langle r,s\rangle \in J$.

Complexity. We ignore the verification in Line 14 and obtain the theorem below, which is proved in Appendix B.

Theorem 3: *Except the costs of verification in Line 14, Alg. 2 needs extra cost of* $O(|\mathcal{U}|\log|\mathcal{U}| + (|\mathcal{R}|+|\mathcal{S}|)l_{avg}(\mathcal{S})) + O(\frac{|\mathcal{R}|\cdot|\mathcal{S}|\cdot l_{avg}(\mathcal{S})}{|\mathcal{U}|}\cdot(1+\frac{|\mathcal{S}|\cdot l_{avg}(\mathcal{S})\cdot w_{sig}}{|\mathcal{U}|^2}))$.

Remark. The former item in Theorem 3 is the total costs to index both input relations, and the latter item is the total costs to perform SCJ. It tells us when FreshJoin performs SCJ efficiently and when inefficiently, which explains well the experimental results in Section 5. In fact, the joining procedure takes (1) nearly constant time when $|\mathcal{U}| >> |\mathcal{R}|\cdot|\mathcal{S}|$; (2)$O(l_{avg}(\mathcal{S}))$ time when $|\mathcal{U}| \approx |\mathcal{R}|\cdot|\mathcal{S}|$; (3)$O(|\mathcal{R}|\cdot l_{avg}(\mathcal{S}))$ time when $|\mathcal{S}| \approx |\mathcal{U}|$; (4)$O(|\mathcal{R}|)$ time when $|\mathcal{S}|\cdot l_{avg}(\mathcal{S}) \approx |\mathcal{U}|$; (5) $O(|\mathcal{S}|\cdot l_{avg}(\mathcal{S})\cdot(1+\frac{|\mathcal{S}|\cdot l_{avg}(\mathcal{S})\cdot w_{sig}}{|\mathcal{U}|^2}))$ time when $|\mathcal{R}| \approx |\mathcal{U}|$; and (6) even $O(|\mathcal{R}|\cdot|\mathcal{S}|^2)$ time when $|\mathcal{U}|$ is very small (comparing to both $|\mathcal{S}|$ and $|\mathcal{R}|$), which is the worst case of FreshJoin.

4. HASH SIGNATURES OF SETS

This section discusses how to generate the hash signatures for all sets adaptively. Section 4.1 presents basic ideas and the framework of our method. Section 4.2 proposes a new hash function. Section 4.3 discusses how to determine signature length w_{sig} with the help of our hash function.

4.1 Framework of Hash Method

As pointed out in [12], frequencies describe many important features of the input datasets, and are useful to make algorithms adaptive. Our hash method distinguishes three kinds of elements in the universe set $\mathcal{U}$, *i.e.,* low frequency elements, mid frequency elements, and high frequency elements. To make the method adaptive to any datasets, the definitions of these elements should not depend on any distribution of the frequencies. Our approach is to fix a constant 0.25 and consider the accumulated frequencies. Notice that, all elements in $\mathcal{U}$ are sorted in increasing order of their frequencies (see Section 2).

Definition 3: Let $total = \sum_{i=1}^{|\mathcal{U}|} f_{\mathcal{S}}(e_i)$. If integers M, H satisfy $\sum_{i=1}^{M-1} f_{\mathcal{S}}(e_i) \leq 0.25\cdot total$, $\sum_{i=1}^{M} f_{\mathcal{S}}(e_i) > 0.25\cdot total$, $\sum_{i=1}^{H-1} f_{\mathcal{S}}(e_i) \leq 0.75\cdot total$, $\sum_{i=1}^{H} f_{\mathcal{S}}(e_i) > 0.75\cdot total$, then all $e_1,...,e_{M-1}$ are called as low frequency elements, all $e_M,...,e_{H-1}$ are called as mid frequency elements, and all $e_H,...,e_{|\mathcal{U}|}$ are called as high frequency elements. □

Although it is very simple, Def. 3 covers many typical cases. For example, if frequencies is distributed uniformly, then the first 25% elements are low frequency ones and the last 25% elements are high frequency ones. However, if frequencies follows a zipfian distribution, then the first $|\mathcal{U}|^{3/4}$ elements are low frequency ones, and the last $|\mathcal{U}|^{1/4}$ elements are high frequency ones.

Example 7: For the universe set of sample relations in Fig. 1, since $f_{\mathcal{S}}(e_i) = i$ $(i \leq 11)$, $total = \sum_{i=1}^{11} i = 66$. We have $\sum_{i=1}^{5} i \leq 0.25\times 66$, $\sum_{i=1}^{6} i > 0.25\times 66$, and $\sum_{i=1}^{9} i \leq 0.75\times 66$, $\sum_{i=1}^{10} i > 0.75\times 66$. Thus, e_1, e_2, e_3, e_4, e_5 are low frequency elements. e_{10}, e_{11} are high frequency elements. And, the others are mid frequency elements. □

The main ideas of our hash method come from the fact that elements with different frequencies are all important to the bitwise filter but for different reasons. In fact, low frequency elements appears in fewer sets, and have sound effects to differ a few sets containing them from many sets not containing them. Similarly, high frequency elements appears in more sets, and have sound effects to differ few sets not containing them from many sets containing them. By contrast, the mid frequency elements also have sound effects to differ many sets containing them from many sets not containing them. Thus, none of these three parts can be ignored, but should be exploited independently.

To do so, all bits in the hash signatures are also partitioned into three parts by two integers M' and H', where $0 < M' < H' < w_{sig}\times 64$. As is illustrated in Fig. 3, all of $b_0, b_1, \cdots, b_{M'-1}$ are used for low frequency elements. Similarly, $b_{M'}, b_{M'+1}, \cdots, b_{H'-1}$ are used for mid frequency elements, and $b_{H'}, b_{H'+1}, \cdots, b_{w_{sig}*64-1}$ are used for high frequency elements. Each bit in each part is used to present whether one or more related elements appear in the given set or not. The mapping between elements and corresponding bits are determined by a hash function named freHash.

Summarily, we get the framework of our hash method (see Alg. 3), where freHash is discussed in Section 4.2. And, the computation of M' and H' is discussed in Section 4.3 together with w_{sig} and some implement issues.

4.2 A New Hash Function

Generally speaking, the framework hashAset can map each element e_i to a bit b_j of the hash signature via arbitrary hash function h. For example, $h(i) = i$ or $h(i) = i\%N$ for

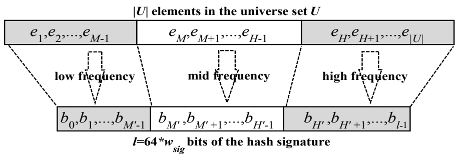


Figure 3: Illustration of basic idea in hash method

```
Algorithm 3: hashAset (aSet)
  Input: A set aSet to be hashed
  Output: the hash signature of the set aSet
1  initialize all bits of $sig$ to be 0;
2  foreach $e_i \in aSet$ do
3      if $i < M$ then                 // low frequency element
4          $j \leftarrow$ freHash $(i)\%M'$;
5      else if $i < H$ then            // mid frequency element
6          $j \leftarrow M'$+freHash $(i-M)\%(H'-M')$;
7      else                            // high frequency element
8          $j \leftarrow H'$+freHash $(i-H)\%(w_{sig}*64-H')$;
9      set $j$-th bit of $sig$ to be 1;
10 return $sig$;
```

a proper N, and so on. In one hand, such functions are helpless in finding a suitable w_{sig} which should be adaptive to input relations. For example, PTSJ [14] just adopted $w_{sig}=\min\{\frac{1}{2}l_{avg}(S), |\mathcal{U}|, 256\}$ heuristically. In the other hand, such functions just take the input i as a usual integer and ignore its important aspect, *i.e.,* i is the inverted rank of e_i's frequency.

Instead, we use a customized hash function, named as freHash. It follows the same ideas from the framework, *i.e.,* low frequency elements, high frequency elements and mid frequency elements are distinguished. The smaller i is, a lower frequency e_i has. Since such e_is have sound effects to differ sets containing such e_is from sets containing none of such e_is, fewer such elements should share a common bit. Similarly, The bigger i is, a higher frequency e_i has. Since such e_is have sound effects to differ sets containing none of such e_is from sets containing such e_is, fewer such elements should share a common bit.

Here comes the formal definition of freHash.

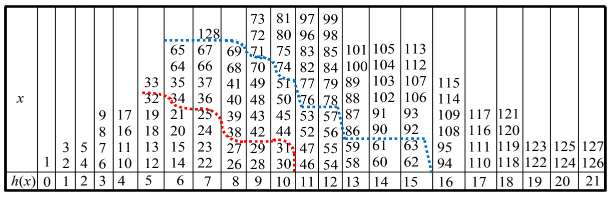
Definition 4: If $i>0$ be an integer and $i=\sum_{k=0}^{\lfloor log_2 i\rfloor} a_k \cdot 2^k$ for $a_k \in \{0,1\}$, then define freHash $(i)=\sum_{k=0}^{\lfloor log_2 i\rfloor} a_k \cdot k$. □

Alg. 4 computes freHash (i) for arbitrary $i>0$. It uses k to trace the position of each bit in the binary representation of i. The 0-th bit is ignored (Line 1). If k-th bit (*i.e.,* a_k) is 1, then value k is accumulated into hash value (Line 3). Then, it considers next bit (Line 4 and Line 5) repeatedly till all bits are considered. Clearly, freHash (i) can be computed in $O(\log i)$ time.

For example, since $23=2^0+2^1+2^2+2^4$, freHash $(23)=1+2+4=7$. Similarly, since $106=2^1+2^3+2^5+2^6$, freHash $(106)=1+3+5+6=15$.

Fig. 4 presents freHash (i) for all $1\leq i\leq 128$. Clearly, as expected, freHash roughly implements the ideas above *i.e.,* fewer elements with both low frequencies or high frequencies have same hash values and more elements with mid frequencies have same values. Moreover, this trend continues when i is in other domains. For instance, in Fig. 4, lower left part under the red dashed line is for $1\leq i\leq 32$, and lower left part under the blue dashed line is for $1\leq i\leq 70$.

```
Algorithm 4: freHash (i)
  Input: an integer $i$
  Output: a hash value of $i$
1  $hashValue \leftarrow 0$; $k \leftarrow 1$; $i \leftarrow i/2$;
2  while $i>0$ do
3      If $i\%2=1$ then $hashValue \leftarrow hashValue+k$;
4      $k \leftarrow k+1$; $i \leftarrow i/2$;
5  return $hashValue$;
```



x	1	3 2	5 4	9 8 7 6	17 16 11 10	33 32 19 18 13 12	65 64 35 34 21 20 15 14	128 67 66 37 36 25 24 23 22	69 68 41 40 39 38 27 26	73 72 71 70 49 48 43 42 29 28	81 80 75 74 51 50 45 44 31 30	97 96 83 82 77 76 53 52 47 46	99 98 85 84 79 78 57 56 55 54	101 100 89 88 87 86 59 58	105 104 103 102 91 90 61 60	113 112 107 106 93 92 63 62	115 114 109 108 95 94	117 116 111 110	121 120 119 118	123 122	125 124	127 126
$h(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 4: Illustration of freHash

An important property of freHash is that freHash (i) is upper bounded if i is. In fact, assume $i\leq n$. Def. 3 tells us that freHash $(i)=\sum_{k=0,a_k\in\{0,1\}}^{\lfloor log_2 i\rfloor} a_k\cdot k \leq \sum_{k=0}^{\lfloor log_2 i\rfloor} k \leq \sum_{k=0}^{\lfloor log_2 n\rfloor} k = \frac{\lfloor\log_2 n\rfloor(\lfloor\log_2 n\rfloor+1)}{2}$.

Definition 5: Let $m_{fh}(n)$ be $\frac{\lfloor\log_2 n\rfloor(\lfloor\log_2 n\rfloor+1)}{2}$ for $n>0$. □

Property 4: *For any* $1\leq i\leq n$, freHash $(i)\leq m_{fh}(n)$.

This means that freHash (i) increases at a rate of logarithmic square and provides a good tool to compute a proper w_{sig} which is adaptive to the datasets.

4.3 Length and Partition of Hash Signatures

Now, we are ready to discuss the signature length w_{sig} and the values M', H' which partition each hash signature into three parts.

The signature length. w_{sig} is taken as the minimum of three values, which give upper bounds of w_{sig} from different points of view.

The first value w_1 gives an upper bound according to the actual needs of freHash. According to Def. 3, e_i is a low (mid or high *resp.*) frequency elements if $i<M$ ($M\leq i<H$ or $H\leq i\leq|\mathcal{U}|$, *resp.*). That is, the input of freHash in Line 4, Line 6 and Line 8 of Alg. 3 is upper bounded by $M-1$, $H-M$, and $|\mathcal{U}|-H+1$ respectively. According to Property 4 (see Section 4.2), hash signatures for low frequency elements, mid frequency elements and high frequency elements need $m_{fh}(M-1)+1$, $m_{fh}(H-M)+1$, and $m_{fh}(|\mathcal{U}|-H+1)+1$ bits, respectively. Thus, the sum of them is the total length of signatures. Thus, we have

$$len(M,H,|\mathcal{U}|) = m_{fh}(M-1)+m_{fh}(H-M)+m_{fh}(|\mathcal{U}|-H+1)$$

$$w_1 = \lceil\frac{1}{64}\cdot(len(M,H,|\mathcal{U}|)+3)\rceil \quad (1)$$

The second value w_2 gives an upper bound according to the average size $l_{avg}(S)$ and the standard deviation $\sigma_{\mathcal{S}}$. View the size of each set as a random variable and apply Chebyshev inequality, we know that $Pr(|s.set| > l_{avg}(S) + 2\cdot\sigma_{\mathcal{S}}) < 0.25$. That is, more than 75% of sets contain at most $l_{avg}(\mathcal{S})+2\cdot\sigma_{\mathcal{S}}$ elements. Thus, we can distinguish these sets from each other by using $l_{avg}(\mathcal{S}) + 2 \cdot \sigma_{\mathcal{S}}$ bit-signature and allowing each bit be reused by different elements. Thus we have

$$w_2 = \lceil \frac{1}{64} \cdot (l_{avg}(\mathcal{S}) + 2 \cdot \sigma_{\mathcal{S}}) \rceil \quad (2)$$

The third value w_3 gives an upper bound according to $|\mathcal{U}|$. Each set can be differed from all others if we use single bit for each element of $\mathcal{U}$. Thus, we have

$$w_3 = \lceil \frac{1}{64} \cdot |\mathcal{U}| \rceil \quad (3)$$

Summarily, in unit of 64-bit integers, the signature length w_{sig} is defined as

$$w_{sig} = \min \{w_1, w_2, w_3\} \quad (4)$$

Partition of Hash Signatures. After w_{sig} is determined, all $64 \cdot w_{sig}$ bits in hash signatures can be partitioned, via integers M' and H', into three parts such that the number of bits in each part is proportional to the actual needs of freHash in mapping elements in each part into bits of the signatures. Thus, we have

$$M' = \lceil 64 w_{sig} \cdot \frac{m_{fh}(M-1)+1}{len(M,H,|\mathcal{U}|)+3} \rceil \quad (5)$$

$$H' = \lceil 64 w_{sig} \cdot \frac{m_{fh}(M-1)+m_{fh}(H-M)+2}{len(M,H,|\mathcal{U}|)+3} \rceil \quad (6)$$

Example 8: For the sample relations in Fig. 1, we cancel the factor $\frac{1}{64}$ in formulas for w_{sig} and get the signature length in bits as follows. Example 7 tells us that M=6 and H=10. Thus, w_1=$m_{fh}(6-1)$+$m_{fh}(10-6)$+$m_{fh}(11-10+1)+3$=8. Since, $l_{avg}(\mathcal{S})$=$\frac{1}{11}\Sigma_{i=1}^{11} i$=6 and $\sigma_S^2 = \frac{1}{11}\Sigma_{i=1}^{11}(i-6)^2$=10, w_2=6+2$\sqrt{10} \approx 12$. And, $w_3 = 11$. Therefore, the signature length is $\min\{w_1, w_2, w_3\}$=8. Assign 8 bits to low, mid, high frequency elements according to their needs, we get M'=3 and H'=6, which results in the SCJ-friendly hash signature in Example 3. □

Remark. After M, H, M', H' is determined, hash value freHash(i) for each element $e_i \in \mathcal{U}$ can be computed and stored in the array unit $HE[i]$ via the operations in Line 3-Line 8 of Alg. 3, which totally takes $|\mathcal{U}| \log |\mathcal{U}|$ time. Then, a single Line "$j = HE[i]$" is substituted for Line 3-Line 8 of Alg. 3, which make the average running time of Alg. 3 be $O(l_{avg}(\mathcal{S}))$. Therefore, the total time to index both datasets, *i.e.,* the costs of Line 1 of Alg. 2, is $O(|\mathcal{U}| \log |\mathcal{U}| + (|\mathcal{R}| + |\mathcal{S}|) l_{avg}(\mathcal{S}))$.

5. EXPERIMENTAL RESULTS

This section empirically evaluates the performance of the proposed techniques via three sets of experiments. The first one checks the adaptivity of FreshJoin. The second one and the third one compare the performance and the scalability of FreshJoin with those of the state-of-the-art algorithms, respectively. All experiments are conducted with single thread on Inspur Server with Intel Xecon 128x2.3GHz CPU and 3TB RAM running CentOS7 Linux.

5.1 Experimental Setup

Algorithms. In all experiments, we compare the following seven algorithms.

- FreshJoin. Our approach proposed in Alg. 2 in Section 3.2. where a freshIndex is built up on $\mathcal{R}$ and $\mathcal{S}$ via Alg. 1 and the hash function proposed in Section 4.
- ttjoin. *Prefix-tree-based* (and *signature-based*) algorithm proposed in [26]. It uses k-least frequent elements in each set of $\mathcal{R}$ as the set's signature. We evaluate ttjoin with k=1,2,$\cdots$,10 and choose the smallest running time in each experiment.
- Limit. *Prefix-tree-based* algorithm proposed in [4], where the depth of the prefix tree is upper bounded with a parameter k. Similarly, We evaluate Limit with k=1,2,$\cdots$,10 and choose the smallest running time.
- PieJoin. *Prefix-tree-based* algorithm proposed in [10], where each prefix tree is stored as several arrays.
- Pretti. *Prefix-tree-based* algorithm proposed in [9].
- Pretti+. *Prefix-tree-based* algorithm proposed in [14], where prefix tree are compressed by contracting the non-branching nodes into single nodes.
- PTSJ. *hash-signature-based* algorithm proposed in [14], where modular is taken as hash function and a PATRICIA Trie is used to help enumerate hash codes.

All these 7 algorithms were implemented in C++ and complied with O3 flag. Among these algorithms, Pretti, Pretti+,PieJoin, and FreshJoin are parameter free. For both ttjoin and Limit, we set k changes from 1 to 10 on each dataset, and choose the smallest running time as results. For PTSJ, we followed the strategy proposed by the authors to take $\min\{|\mathcal{U}|, \frac{1}{2} \cdot l_{avg}(S), 256\} \times 64$ as the signature length (in bits). As shown in [10, 26], the order of elements in sets had a huge impact for Limit, PieJoin and pretti+. Thus, we also followed their empirical conclusion to apply infrequent sort order for Limit, PieJoin, and Pretti, and frequent order for pretti+. For ttjoin, the infrequent order is applied on $\mathcal{R}$, and the frequent order is applied on $\mathcal{S}$. While for FreshJoin, the infrequent order is applied on both inputs.

As in literatures, all algorithms were run to join each selected dataset with itself.

Dataset. We adopt 16 real-life datasets selected from different domains with various data properties. The detailed characteristics of these datasets are shown in Table 2. For each dataset, we showed the type of the dataset, what the sets and elements represent, the number of sets in the dataset, the average set length and the number of elements in the universe. Half of these datasets are same as in [26]. Other datasets are different, because we obtain them from KONECT[1] (rather than the addresses given in the literatures) and extract all sets in each datasets via the tools provided there or the approaches described in literatures. Such datasets include Discogs, AOL, Enron and OrKut. Particularly, to obtain datasets with bigger average set length to check the adaptivity of FreshJoin, we modify the extracting approaches for Webbase and Netflix. In Webbase, we take pages as elements and extract all pages with the number of outlinks no smaller than 2500 as records. While in Netflix, we take movies as records and audiences as elements. All datasets are sorted in lexicographical order of sets before they are fed into the algorithms.

[1]http://konect.uni-koblenz.de/

Table 2: Characteristic of Real Datasets

Dataset	**Abbrev.**	**Type**	**Set**	**Element**	$\|\mathcal{S}\|$	$l_{avg}(\mathcal{S})$	$\|\mathcal{U}\|$	M	H	w_{sig}	M'	H'
Linux[26]	LINUX	Interaction	Thread	User	337,509	1.78	42,045	41,448	42,015	1	43	52
Stack [26]	STACK	Rating	User	Post	545,196	2.39	96,680	81,551	95,585	1	31	51
Discogs[26]	DISCO	Affiliation	Artist	Label	7,991,155	2.40	7,949,791	4,682,322	7,840,873	1	26	50
Bookcrossing[26]	BOOKC	Rate	Book	User	337,578	3.40	105,091	98,953	104,894	1	35	56
Amazon[26]	AMAZ	Rating	product	User	1,231,019	4.67	2,146,277	1,436,024	2,133,860	1	28	52
BMS[4]	BMS	Sale	Transaction	Product	515,597	6.53	1,657	1550	1650	1	43	56
Kosarak[4]	KOSRK	Interaction	User	Link	990,002	8.10	41,270	39,789	41,263	1	42	56
Delicious[26]	DELIC	Folksonomy	User	Tag	666,841	11.87	685,563	647,962	685,362	1	36	56
AOL[26]	AOL	Text	Query	Keyword	657,427	26.09	10,154,742	4,287,838	10,115,374	1	26	52
Twitter[14]	TWIT	Interaction	Partition	User	456,626	32.53	370,341	338582	369740	1	34	55
LiveJournal[26]	LIVEJ	Affiliation	User	Group	3,201,203	35.08	7,489,296	7,456,367	7,488,933	1	41	56
OrKut[14]	ORKUT	Interaction	User	Community	3,072,589	38.14	3,072,626	1,962,178	2,932,062	1	25	47
Enron[26]	ENRON	Text	Email	word	516,782	111.49	435,261	430,538	435,085	3	116	171
Reuters	REUTRS	Text	Word	story	283,911	213.34	781,265	404,447	706,074	2	46	92
Webbase[14]	WEBBS	Web	Page	outlink	168,704	2,976	6,142,611	5,881,138	6,121,663	2	63	101
Netflix[4]	NETFX	Rating	Movie	Audience	17,770	5,654	480,189	340,724	460,135	4	106	190

5.2 Experimental Results

Exp1: Adaptivity. To evaluate the adaptivity of FreshJoin, we ran FreshJoin on all 16 datasets, and recorded values of M, H, $M\prime$, H' and w_{sig} (see Table 1 for their meaning) respectively. The results are reported in the last column of Table 2.

We find that (1) the partitions of elements in universes into low, mid, and high frequency elements are adaptive to the dataset themselves. Even more, if we assume all datasets follow Zipfian distribution(see experiments of [26]), then the more skewed the dataset is the bigger M, H we have; (2) the lengths of hash signatures, which are determined by Formula 4 in Section 4.3, always keep reasonably small and also change adaptively; And (3) the splits of hash signatures into three parts, which is determined by Formula 5 and Formula 6 in Section 4.3, also change adaptively. Therefore, FreshJoin is a parameter-free adaptive algorithm.

Exp2: Performance. This set of experiments compares our FreshJoin algorithm with 6 state-of-the-art algorithms ttjoin, PieJoin, Pretti, Pretti+, Limit, and PTSJ on all 16 datasets. On each dataset, we recorded the space costs of each algorithm and its the total running time which include the time to index the dataset and the time to join the dataset. The results are reported in Fig. 5 and Fig. 6 respectively. Besides, for FreshJoin, we also recorded the time to compute the statistics of each dataset. These results are not reported, because these costs are very small (≤ 0.03 seconds on all datasets) compared to the total running times.

For memory usage (see Fig. 5), we find that (1)Except FreshJoin, Pretti+ and Limit always uses less memory, while ttjoin and PieJoin need more memory. This indicates that the prefix trees are space expensive and the compressing strategies proposed in [4, 14] are effective. (2)FreshJoin almost always uses least main memory, except on Linux and AOL which is the worst cases of FreshJoin according to the remark at the end of Section 3.1 but the costs of FreshJoin are still competitive to the costs of other algorithms. Moreover, FreshJoin saves nearly 50% space of LIMIT and Pretti+ and more than 70% of ttjoin and PTSJ on datasets such as Twitter,LiveJournal, OrKut,Enron, Reuters, Webbase,NetFlix, while keeps competitive on other datasets. This behavior verified the space efficiency of our sparse index structures and the cost analysis in Theorem 2. (3)FreshJoin uses much less space than PTSJ. This verified the effectiveness of Formula 4 in Section 4.3.

For processing time (see Fig. 6), we find that (1)FreshJoin is faster than all other algorithms on more than half of datasets, which benefits from the efficient index structure and the joining procedure of FreshJoin. (2)FreshJoin is a little bit slower (but still competitive to) than some of ttjoin, PieJoin, Pretti, Pretti+,Limit or even PTSJ on some datasets, where $|\mathcal{U}| \ll |\mathcal{S}|$ holds. According to Theorem 3, these are the worst cases for FreshJoin. (3)FreshJoin is always faster than PTSJ, except on the worst case of BMS. These observations verified the effective and efficiency of our adaptive algorithm and the correctness of our analysis of the time complexity of FreshJoin. Therefore, Theorem 3 provides us a rule to choose FreshJoin from the existing set containment algorithms.

Exp3: Scalability. The third set of experiments compares the scalability of 7 algorithms on 4 representative datasets. We choose Amazon, OrKut, NetFlix and Webbase as datasets here, which have different average set lengths $l_{avg}(S)$. Similar in [26], we randomly sampled 20%, 40%, 60%, 80%, and 100% of sets from each dataset, and conducted experiment on each sampled datasets. The total running time and space costs of each algorithm are recorded. The results are reported in Fig. 7 and Fig. 8 respectively. We find that both the total running times and the space costs of FreshJoin grows slowly and steadily as $|\mathcal{S}|$ increases.

Summary. Experiments on 16 real-life datasets show that our parameter-free hash-signature-based set containment join algorithm is adaptive, well scaled, efficient and effective. According to Theorem 3, FreshJoin performs SCJ efficiently both in space and time if $|\mathcal{U}| \ll |\mathcal{S}|$ is not the case.

6. RELATED WORK

Bulk comparison of sets has many practical applications in various domains such as graph analytical tasks, query optimization, OLAP and data mining [14]. Therefore, people have studied extensively the theory and engineering of different operations involving set comparison such as containment queries [1, 8, 11, 19, 20, 25, 27], similarity joins [2, 3, 5, 6, 21, 22, 23, 24], equality joins [15] and containment joins [4, 7, 9, 10, 14, 15, 16, 17, 18, 26].

Early work on set containment join mainly focused on disk-based algorithms (*e.g.,* [7, 15, 17, 18]). Although these

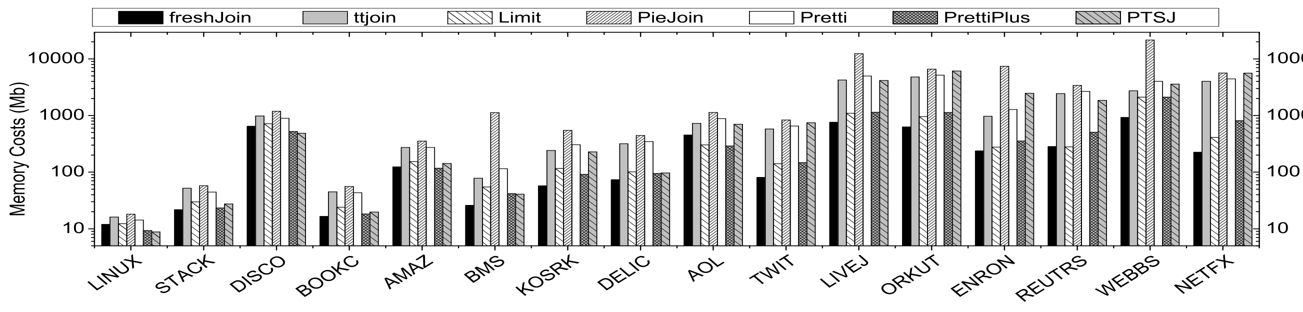


Figure 5: The memory costs of different algorithms

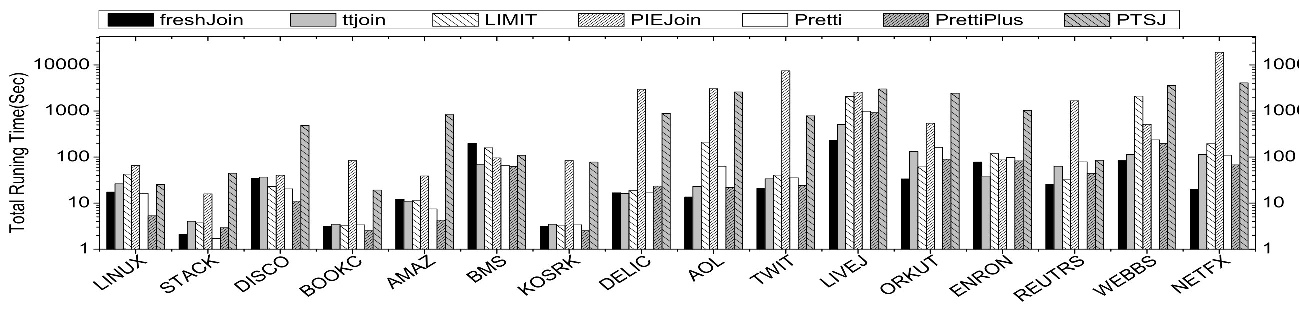
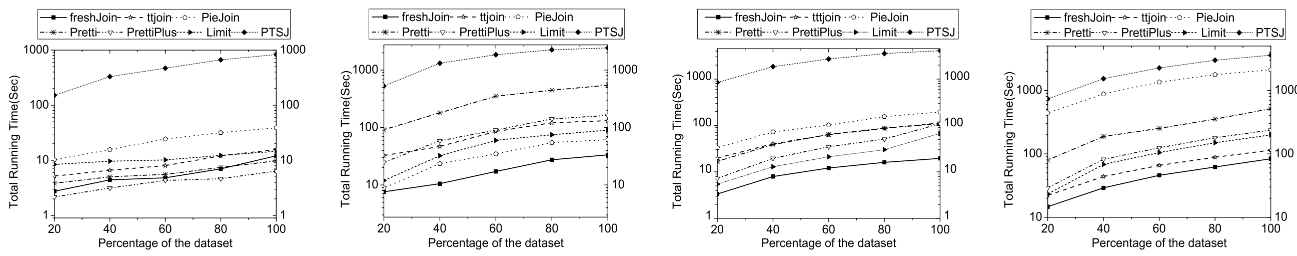
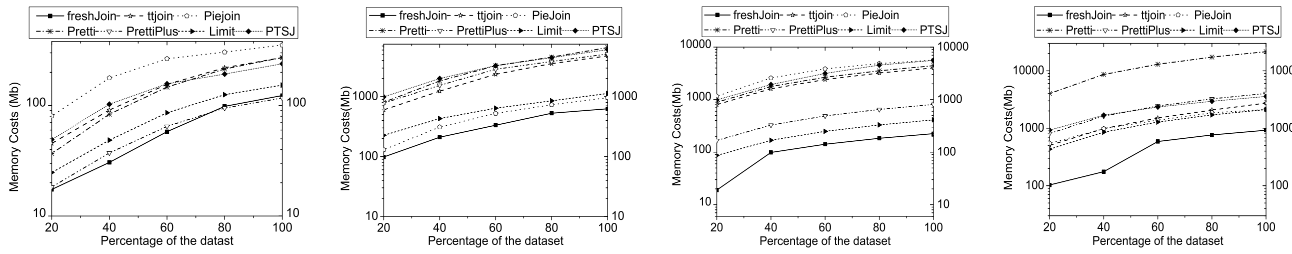


Figure 6: The total running time of different algorithms



(a) Amazon (b) Orkut (c) NetFlix (d) Webbase

Figure 7: Scalability in total running time



(a) Amazon (b) Orkut (c) NetFlix (d) Webbase

Figure 8: Scalability in memory costs

algorithms have proven quite effective for joining massive set collections, their performance is bounded by their underlying in-memory processing strategies [14]. For example, PSJ [18] and APSJ [17], two advanced disk-based algorithms, share the same in-memory processing strategy with SHJ [7]. To keep up with ever-increasing data volumes and modern hardware trends, recent work turn to develop next generation in-memory SCJ algorithms [7, 4, 9, 10, 14, 26], which are either signature-based or prefix-tree based.

All signature-based algorithms (e.g.,SHJ [7], PSJ [18], APSJ [17] and PTSJ [14]) follow the filter-and-refine framework in Section 2. They use fixed-length bitmaps as signatures to approximate sets, and adopt bitwise operation on the signatures as a filter to prune away as many as possible tuple-pairs whose sets do not have subset relationship. All these existing algorithms take different empirical values as the lengths of bitmaps, and use traditional rand function or element modulo bitmap length as hash functions. This makes them hardly adaptive to datasets automatically. Instead, they care about how to find potential signatures pairs that may pass through the bitwise filter. The usual way is to, for each signature from $\mathcal{R}$, enumerate all potential signatures from $\mathcal{S}$, which incurs high CPU costs and works only on short signatures although special structures such as PATRICIA

TRIE [14] can be used to alleviate this defect in some extent. Compare to them, our method computes the length of signature adaptively and totally avoids enumerating signatures by exploiting the least frequent elements in sets to associate signature-pairs which are fed to the bitwise filter.

Most prefix-tree-based algorithms (*e.g.,*Pretti [9], Pretti+ [14], LIMIT [4], Piejoin [10]) build a prefix tree $T_{\mathcal{R}}$ and create an inverted index I on $\mathcal{S}$, where $I_{\mathcal{S}}(e_i)$ records all $s \in \mathcal{S}$ with $e_i \in s.set$. Then, they traverse $T_{\mathcal{R}}$ in depth-first manner to visit each set $r.set(r \in \mathcal{R})$, compute the intersection $\cap_{e_i \in r} I_{\mathcal{S}}(e_i)$ at the same time, and output $\langle r, s \rangle$ for each s in the intersection. Since common prefix of different sets is represented as a common path in prefix tree, thus many partial results of the intersection can be shared by many tuples in $\mathcal{R}$. Notice that, prefix tree are space-costly and traverses of deep paths in the tree are time-costly. So many optimization techniques are adopted. For example, LIMIT [4] limited the height of tree empirically, Pretti+ [14] compressed the prefix tree by merging these non-branching nodes along each path into single nodes, and Piejoin [10] transforms the prefix trees into linear arrays via preorder coding.

The state-of-the-art algorithm ttjoin [26] are based on both signatures and prefix trees. It takes k-least frequent elements of sets in $\mathcal{R}$ as their signatures and indexes signatures in a prefix tree $T_{\mathcal{R}}$. Besides, all sets in $\mathcal{S}$ are indexed in an other prefix tree $T_{\mathcal{S}}$. Then, ttjoin traverses $T_{\mathcal{S}}$ depth-firstly to visit each set s of $\mathcal{S}$. When each node n of $T_{\mathcal{S}}$ is visited, ttjoin obtains the label e of n and checks whether e is the least frequent element of a set in $\mathcal{R}$ by traversing $T_{\mathcal{R}}$ in depth-first manner. Again, the empirical parameter k makes ttjoin not adaptive to datasets automatically. Besides, the prefix tree $T_{\mathcal{S}}$ is space costly.

As is pointed out in [26], some existing set similarity search algorithms (*e.g.,* [1], [11] and [22]) can be adapted to support set containment join by setting specific thresholds. For example, using a nested loop and setting T in the generalized T-occurrence query as the size of *r.set* for each $r \in \mathcal{R}$, DivideSkip [11] can also be extended to compute set containment join. Similarly, by setting setting the overlap threshold T to be the size of *r.set* for each $r \in \mathcal{R}$ in the nested-loop procedure, the adaptive framework for set similarity search proposed in [22] can also be utilized to compute set containment join. By setting the error-tolerate threshold as 1 in the index structure for error-tolerant set containment search, the algorithm in [1] can be applied to support set containment join. However, as is shown in the experimental result of ttjoin [26], these renewed algorithms are not as competitive as those SCJ-specific algorithms.

While statistics have been adopted in query optimizations (*e.g.,* [13]), they are not widely used (except) to accelerate set containment join algorithms, to the best of our knowledge. Besides, filter-and-refine frameworks have been widely used in string similarity join (*e.g.,* [12]), they are not widely used in set containment join, except in SHJ [7], PTSJ [14], APSJ [17] and ttjoin [26]).

7. CONCLUSION

This paper revisits the set containment join problem and proposes a parameter-free hash-signature-based join algorithm. It exploits the frequencies of elements to partition the universe set into low, mid, and high frequency elements, and maps them separately into different parts of the hash-signature of each record via a new proposed hash function, which also provides a tool to adaptively estimate the length of hash signatures. The hash signatures are well organized into a index. The time and space complexities of the algorithm are analyzed. Experiments on 16 real-life datasets indicate that the proposed algorithm is adaptive, well scaled, efficient and effective.

8. REFERENCES

[1] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, pages 927–938, 2010.

[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, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, pages 1–28, 2015.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[6] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. In *VLDB*, pages 360–371, 2015.

[7] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison predicates. In *VLDB*, pages 386–395, 1997.

[8] Z. Hmedeh, H. Kourdounakis, V. Christophides, M. S. C. Du Mouza, and N. Travers. Subscription indexes for web syndication systems. In *EDBT*, pages 312–323, 2012.

[9] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *DASFAA*, pages 761–772, 2005.

[10] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros, and U. Leser. Piejoin: Towards parallel set containment joins. In *SSDBM*, pages 11–22, 2016.

[11] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[12] J. Luo, S. Shi, H. Wang, and J. Li. Frepjoin: an efficient partition-based algorithm for edit similarity join. *Frontiers of Information Technology & Electronic Engineering.*, 18(10):1499–1510, 2017.

[13] J. Luo, X. Zhou, Y. Zhang, H. Shen, and J. Li. Selectivity estimation by batch-query based histogram and parametric method. In *Australia Database Conference*, pages 93–102, 2007.

[14] Y. Luo, G. H. Fletcher, J. Hidders, and P. D. Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *ICDE*, pages 303–314, 2015.

[15] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, pages 157–168, 2003.

[16] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *EDBT*, pages 427–444, 2002.

[17] S. Melnik and H. G. Molina. Adaptive algorithms for set containment joins. *TODS*, 28(1):56–99, 2003.

[18] 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.

[19] M. Terrovitis, P. Bouros, P. Vassiliadis, T. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *EDBT*, pages 225–236, 2011.

[20] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737, 2006.

[21] N. A. W. Mann and P. Bouros. An empirical evaluation of set similarity join techniques. In *VLDB*, pages 636–647, 2016.

[22] 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.

[23] C. Xiao, X. L. W. Wang, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2008.

[24] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[25] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *TODS*, 19(2):332–364, 1994.

[26] J. Yang, W. Zhang, S. Yang, Y. Zhang, and X. Lin. Tt-join: Efficient set containment join. In *ICDE*, pages 509–520, 2017.

[27] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. Lsh ensemble: Internet scale domain search. In *VLDB*, pages 1185–1196, 2016.

APPENDIX

A. PROOF OF LEMMA 1

Proof: Let u be the size of $\mathcal{U}$,*i.e.,* $\mathcal{U}=\{e_1, e_2, \cdots, e_u\}$.

First, we assert that the probability of e_i being marked by an arbitrary tuple $r \in \mathcal{R}$ is $\frac{1}{u}(1-\frac{1}{u})^{i-1}+\frac{i-1}{u^2}(1-\frac{1}{u})^{i-2}$, *i.e.,* $Pr(e_1^{(r)} = e_i \vee e_2^{(r)} = e_i)=\frac{1}{u}(1-\frac{1}{u})^{i-1}+\frac{i-1}{u^2}(1-\frac{1}{u})^{i-2}$. In fact, since *r.set* is chosen from $\mathcal{U}$ uniformly, the probability of each element being chosen is $\frac{1}{u}$. Therefore, the assertion follows from the facts below. $e_1^{(r)} = e_i$ means that e_i is in *r.set* but none of $e_1, \cdots, e_{i-1}$ is in *r.set*. Similarly, $e_2^{(r)} = e_i$ means that e_i is in *r.set* and only one of $e_1, \cdots, e_{i-1}$ is in *r.set*.

Next, we estimate the probability of e_i being marked by Alg. 1, *i.e., $Pr(e_i$ is marked$)$*. Notice that, e_i being marked means it being marked by at least one tuple of $\mathcal{R}$. Since each set *r.set* ($r \in \mathcal{R}$) is chosen independently, $Pr(e_i \text{ is marked}) \le |\mathcal{R}| \cdot Pr(e_1^{(r)} = e_i \vee e_2^{(r)} = e_i)$.

Now, Let $X_i = 1$ if e_i is marked by Alg. 1, and $X_i = 0$ otherwise. Therefore, $X = \sum_{i=1}^{u} X_i$ is the total number of marked elements. The lemma follows from the computation below.

$$\begin{aligned}
E[X] &= \sum_{i=1}^{u} E[X_i] \\
&= \sum_{i=1}^{u} Pr(e_i \text{ is marked}) \\
&\le \sum_{i=1}^{u} |\mathcal{R}| \cdot Pr(e_1^{(r)} = e_i \vee e_2^{(r)} = e_i) \\
&= |\mathcal{R}| \cdot [\sum_{i=1}^{u} \frac{1}{u}(1-\frac{1}{u})^{i-1} + \sum_{i=2}^{u} \frac{i-1}{u^2}(1-\frac{1}{u})^{i-2}] \\
&= |\mathcal{R}| \cdot \{[1-(1-\frac{1}{u})^u] + [1 - \frac{2u-1}{u-1}(1-\frac{1}{u})^u]\} \\
&= |\mathcal{R}| \cdot [2 - \frac{3u-2}{u-1}(1-\frac{1}{u})^u] \\
&= |\mathcal{R}| \cdot [2 - 3(1-\frac{1}{u})^{u-1} + \frac{2}{u-1}(1-\frac{1}{u})^u] \\
&\le (2 - \frac{3}{\mathbf{e}} + \frac{2}{(u-1)\mathbf{e}}) \cdot |\mathcal{R}| \\
&\approx 0.9 \cdot |\mathcal{R}|
\end{aligned}$$

The last inequity follows from the fact that $(1-\frac{1}{u})^{u-1} > \frac{1}{\mathbf{e}}$ and $(1-\frac{1}{u})^u < \frac{1}{\mathbf{e}}$ hold for any integer $u \ge 2$. $\square$

B. PROOF OF THEOREM 3

Proof: According to the remark in Section 4.3, the time costs to index $\mathcal{R}$ and $\mathcal{S}$ (*i.e.,* Line 1) is $O(|\mathcal{U}| \log |\mathcal{U}| + (|\mathcal{R}| + |\mathcal{S}|) l_{avg}(\mathcal{S}))$. Thus, we only need to analyze the costs for Line 2-Line 13.

First, we analyze the total costs of Line 9. Since each set *s.set* ($s \in \mathcal{S}$) contains $l_{avg}(\mathcal{S})$ elements on average, there are total $|\mathcal{S}| \cdot l_{avg}(\mathcal{S})$ items in the inverted index of $\mathcal{S}$. Therefore, each inverted list contains $l = \frac{|\mathcal{S}| \cdot l_{avg}(\mathcal{S})}{|\mathcal{U}|}$ items on average. It follows that Line 9 needs $2l$ comparisons each time when it is executed. In the worst case, Line 9 is executed for each item in the inverted invert of $\mathcal{R}$. Since each tuple of $\mathcal{R}$ is indexed only once, the total costs of Line 9 is $\sum_{r \in \mathcal{R}} 2l = 2 \cdot \frac{|\mathcal{R}||\mathcal{S}| l_{avg}(\mathcal{S})}{|\mathcal{U}|}$.

Then, we analyze the average length of $List = I_{\mathcal{S}}(e_i) \cap I_{\mathcal{S}}(e_u)$ after Line 9 is executed. For each $j \in I_{\mathcal{S}}(e_i)$, we know $e_i \in s_j.set$. If $j \in I_{\mathcal{S}}(e_u)$, then one of the other $l-1$ elements in $s_j.set$ must be e_u, which happens with probability $\frac{l-1}{|\mathcal{U}|}$. Since there are l such js in $I_{\mathcal{S}}(e_i)$ on average, the average length of $List$ is $\frac{l(l-1)}{|\mathcal{U}|} \le \frac{l^2}{|\mathcal{U}|}$.

Now, we are ready to analyze the total costs of Line 12. First of all, Line 12 needs in worst case $2w_{sig}$ comparisions each time when it is executed. In fact, since the hash code contains w_{sig} 64-bit integers, the bitwise AND operation between each pair of integers costs 1 and the comparison ($\neq$) costs 1. Since there are $\frac{l(l-1)}{|\mathcal{U}|}$ tuples indexed in $List$ after Line 9 is executed, each run of Line 11 and Line 12 needs $\frac{l(l-1)}{|\mathcal{U}|} \cdot 2w_{sig}$ time. Finally, since Line 11 and Line 12 will be executed for each $r \in \mathcal{R}$, the total costs of Line 12 is $\frac{l(l-1)}{|\mathcal{U}|} \cdot 2w_{sig} \cdot |\mathcal{R}|$.

Summarily, Line 2-Line 13 of Alg. 2 need the total cost below, as is expected in the theorem.

$$\begin{aligned}
cost_{extra} &= cost(Line11) + cost(Line12) \\
&= 2\frac{|\mathcal{R}||\mathcal{S}| l_{avg}(\mathcal{S})}{|\mathcal{U}|} + \frac{l(l-1)}{|\mathcal{U}|} \cdot 2w_{sig} \cdot |\mathcal{R}| \\
&\le 2\frac{|\mathcal{R}||\mathcal{S}| l_{avg}(\mathcal{S})}{|\mathcal{U}|} \cdot (1 + \frac{|\mathcal{S}| l_{avg}(\mathcal{S}) w_{sig}}{|\mathcal{U}|^2})
\end{aligned}$$