Data Science and Engineering (2019) 4:293–308
https://doi.org/10.1007/s41019-019-00107-y

# FreshJoin: An Efficient and Adaptive Algorithm for Set Containment Join

**Jizhou Luo[1] · Wei Zhang[1] · Shengfei Shi[1] · Hong Gao[1] · Jianzhong Li[1] · Wei Wu[1] · Shouxu Jiang[1]**

Received: 14 August 2019 / Revised: 16 October 2019 / Accepted: 22 October 2019 / Published online: 9 November 2019


**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 cost because of the enumeration of signatures, while the latter incurs high space cost 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 builds a flat index on-the-fly to record three kinds of signatures (i.e., two least frequent elements and a hash signature whose length is determined adaptively by the frequencies of elements in the universe set). The index consists of two sparse inverted indices and two arrays which record hash signatures of all sets in each relation. The index is well organized such that FreshJoin can avoid enumerating hash signatures. The rationality of this design is explained. And, the time and space cost of the proposed algorithm, which provide a rule to choose FreshJoin from existing algorithms, are analyzed. Experiments on 16 real-life datasets show that FreshJoin usually reduces more than 50% of space cost while remains as competitive as the state-of-the-art algorithms in running time.

**Keywords** Set containment join · Frequency hash · Join algorithm

## 1 Introduction

Sets are ubiquitous and widely used in databases, where data are processed and analyzed. For example, a set-valued attribute of a tuple may record the prerequisites of a course, or the labels of a digital image, or the tokens in an email, and so on. With this comes a large body of research interests on efficient algorithms for fundamental operations on such attributes such as containment joins [1–11], containment queries (e.g., [12]), and similarity joins (e.g., [13]).

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. Figure 1a illustrates prerequisites of courses in $\mathcal{R}$, and Fig. 1b 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 [1–11]. Among these, the early SCJ algorithms [7–11] are mainly disk-based. And, their performances are mainly bounded by their underlying in-memory processing strategies [4]. Recently, researchers

✉ Jizhou Luo
luojizhou@hit.edu.cn

Wei Zhang
weizhang@hit.edu.cn

Shengfei Shi
shengfei@hit.edu.cn

Hong Gao
honggao@hit.edu.cn

Jianzhong Li
lijz@hit.edu.cn

Wei Wu
weiwu@hit.edu.cn

Shouxu Jiang
jsx@hit.edu.cn

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China

**Fig. 1** Two relations for running examples

| 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 |
|---|---|
| $s_1$ | $\{e_1,e_3,e_5,e_6,e_9,e_{11}\}$ |
| $s_2$ | $\{e_2,e_4,e_5,e_9,e_{10},e_{11}\}$ |
| $s_3$ | $\{e_2,e_5,e_7,e_9,e_{10},e_{11}\}$ |
| $s_4$ | $\{e_3,e_7,e_8,e_9,e_{10},e_{11}\}$ |
| $s_5$ | $\{e_3,e_8,e_9,e_{10},e_{11}\}$ |
| $s_6$ | $\{e_4,e_5,e_6,e_7,e_8,e_9\}$ |

| id | set |
|---|---|
| $s_7$ | $\{e_4,e_6,e_7,e_{10},e_{11}\}$ |
| $s_8$ | $\{e_4,e_7,e_8,e_{10},e_{11}\}$ |
| $s_9$ | $\{e_5,e_6,e_8,e_9,e_{10},e_{11}\}$ |
| $s_{10}$ | $\{e_6,e_7,e_8,e_{10},e_{11}\}$ |
| $s_{11}$ | $\{e_6,e_8,e_9,e_{10},e_{11}\}$ |
| $s_{12}$ | $\{e_7,e_8,e_9,e_{10},e_{11}\}$ |

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

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

turned to study in-memory algorithms [1–6], due to the improvement in modern hardware and the popularity of distributed computing infrastructures. Such algorithms are either signature-based or prefix-tree-based.

Signature-based algorithms (e.g., SHJ [11], PSJ [10], APSJ [8], and PTSJ [4]) encode each set as a hash signature and use bitwise operation on signatures as a filter to check possible containment. The main challenge there is how to find potential signature pairs that may pass through the filter. The usual way is to, for each signature from $\mathcal{R}$, enumerate all potential signatures from $\mathcal{S}$, which incurs high CPU cost and works well only on short signatures although special structures such as PATRICIA TRIE [4] can be used to alleviate this defect to some extent.

Prefix-tree-based algorithms (e.g., Pretti [6], Pretti+ [4], LIMIT [5], Piejoin [3], ttjoin [2], LCJoin [1]) achieve high speeds by exploiting prefix tree(s) to share the intersection operations (of inverted lists) among any tuples $r \in \mathcal{R}$ which have common prefixes. However, prefix trees need high space cost. In fact, when the average set size is large, the space for prefix trees is several times of space for the data itself. Although the space cost can be sharply reduced by limiting the height of the tree (e.g., LIMIT [5], ttjoin [2]), or storing the tree as several arrays (e.g., Piejoin [3]), or compressing the non-branching nodes into single nodes (Pretti+ [4]), these algorithms do not perform well on all datasets because their performance depends on some empirical parameters (e.g., the height of prefix trees) or other factors which are hardly adaptive to datasets themselves.

In the big data era, it is important to make SCJ algorithms well scaled in both space cost 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 cost. Instead, it creates a flat index on-the-fly to record three kinds of signatures. Arrays are used to store the hash signatures (i.e., bitmaps) of sets in both $\mathcal{R}$ and $\mathcal{S}$, and inverted indices are designed to store two least frequent elements of sets in $\mathcal{R}$ and elements of sets in $\mathcal{S}$. These signatures are well organized such that (1) FreshJoin can use the bitwise filter (like in SHJ [11] and PTSJ [4]) but without enumerating the hash signatures; 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. Besides, FreshJoin performs SCJ adaptively according to the statistics of the datasets by allowing the lengths of hash codes to change adaptively. Compared with the state-of-the-art SCJ algorithms, FreshJoin usually keeps as competitive as its counterparts in running time and reduces even more than 50% of space cost. In the worst case, it remains as competitive as its counterparts in both space cost and running time. Our theoretical analysis provides a rule to distinguish the worst case from other cases.

Our main contributions include: (1) We propose a parameter-free adaptive algorithm to evaluate SCJ efficiently, (2) we propose a sparse asymmetric inverted index to make three kinds of signatures work coordinately and economically, (3) we propose a new hash function to estimate the signature length adaptively by partitioning the elements into three groups according to their frequencies, and (4) we conduct experiments on 16 real-life representative datasets and find that our algorithm is adaptive, well scaled and efficient.

This paper extends an earlier version [14] with the following added value. A new section, named as preliminaries, is added to make it easier for readers to focus on the main contributions of this article. Lemmas and theorems in [14] are all proved carefully in Sect. 3. The rationality of partitioning the universe set into three parts is explained in Sect. 4. The description of the experiments is reorganized, and scalability experiment is added in Sect. 5. Besides, more related works are compared in Sect. 6.

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

**Table 1** 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 mid frequency |

## 2 Preliminaries

This section introduces basic concepts and definitions. Table 1 summarizes the important notations used throughout this article.

We assume a discrete universe set, denoted as $\mathcal{U}$, consisting of a linearly ordered list of elements $e_1, e_2, \ldots, e_n$. $n$ is called as the size of $\mathcal{U}$, denoted as $n = |\mathcal{U}|$. A subsequence $e_{k_1}, e_{k_2}, \ldots, e_{k_m}$ of $e_1, e_2, \ldots, e_n$ 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 $A, B$, ..., i.e., $A = \{e_{k_1}, e_{k_2}, \ldots, e_{k_m}\}$ means $A$ is the set $e_{k_1}, e_{k_2}, \ldots, e_{k_m}$. $e_i \in A$ 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 $\mathbb{R} = (\mathbb{A}_1, \ldots, \mathbb{A}_p, SET)$, where $\mathbb{A}_i$ is an attribute with domain $\Omega_i$ for $i = 1, \ldots, p$ 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 $SET$ a 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. 1a) and $\mathcal{S}$ (Fig. 1b), the result of SCJ $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$ 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 an 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. 1a) and $\mathcal{S}$ (Fig. 1b), element $e_i$ in the shared universe $\mathcal{U} = \{e_1, e_2, \ldots, e_{11}\}$ has $\mathcal{S}$-frequencies $i$ for $i = 1, 2, \ldots, 11$. And, $\mathcal{U}$ is sorted in an 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 [6], Pretti+ [4], LIMIT [5], Piejoin [3], and ttjoin [2]) benefit from this by accelerating both the creation of prefix trees and the joining procedures. This paper also assumes the input relations be sorted in this way. For instance, $\mathcal{R}$ (Fig. 1a) and $\mathcal{S}$ (Fig. 1b) 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 $O(|\mathcal{R}| \cdot |\mathcal{S}| \cdot l_{avg}(S))$ 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 $r.set$ not being a subset of $s.set$ 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 [11] and PTSJ [4], 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

We first describe our index structure and discuss its creation and space cost (Sect. 3.1). Then, we present FreshJoin algorithm, state its correctness, and analyze its complexity (Sect. 3.2). The details of the hash function are postponed to the next section, except that the signature length $w_{sig}$ is assumed.

### 3.1 The Index and Its Creation

The FreshJoin algorithm uses three kinds of signatures. The first is the hash codes associated with tuples in both $\mathcal{R}$ and $\mathcal{S}$. The second is the least frequent element in set $r.set$ for each $r \in \mathcal{R}$. And, the third is the second least frequent elements in set $r.set$ for each $r \in \mathcal{R}$. 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 hash codes for $\mathcal{R}$ and $\mathcal{S}$, respectively. Each unit of the arrays stores $w_{sig}$ 64-bit integers and can be accessed via the IDs of tuples in $\mathcal{R}$ or $\mathcal{S}$.

A sparse inverted index $I_{\mathcal{R}}$ is used to index the second and third kind of signatures for tuples in $\mathcal{R}$. Each $e \in \mathcal{U}$ 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 are sorted such that items with a same second component are stored contiguously. In this way, all tuples $r$ with $r.set$ having the same second 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.

Besides, an other sparse inverted index $I_{\mathcal{S}}$ is used to index the tuples in $\mathcal{S}$. Each $e \in \mathcal{U}$ 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 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. Particularly, if $e \in \mathcal{U}$ is not the second or third type of signatures for any tuple $r \in \mathcal{R}$, then $I_{\mathcal{S}}(e)$ is null.

***Example 4*** Figure 2 illustrates the index with data in Fig. 1. Array $sig_{\mathcal{R}}$ ($sig_{\mathcal{S}}$ *resp.*) stores hash signatures of tuples in $\mathcal{R}$ ($\mathcal{S}$ *resp.*), which are generated via the toy hash function in Example 3. In $I_{\mathcal{R}}$, $I_{\mathcal{R}}(e_1)$ 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 $I_{\mathcal{R}}$ are similar. While in $I_{\mathcal{S}}$, $I_{\mathcal{S}}(e)$ is non-null only if $e$ is one of the first two least elements of a set $r_i.set$. For instance, $I_{\mathcal{S}}(e_8)$ contains

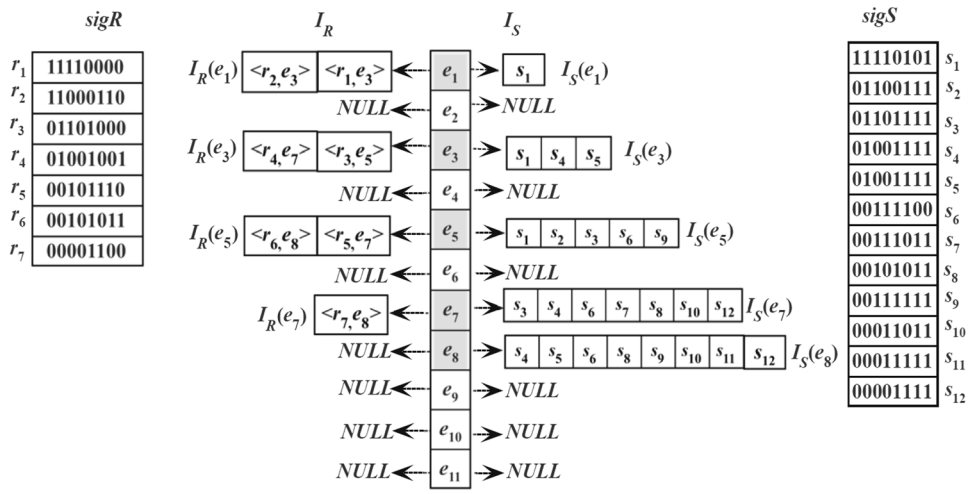



**Fig. 2** freshIndex of sample relations in Fig. 1

eight items. $I_S(e_{11})$ is null, although $e_{11}$ is the most frequent element.

*The Creation of freshIndex* The idea to create freshIndex is rather simple. Both $\mathcal{R}$ and $\mathcal{S}$ are sorted lexicographically first, which guarantees a natural order on inverted lists in both $I_\mathcal{R}$ and $I_\mathcal{S}$. Then, $\mathcal{R}$ is indexed and elements in $\mathcal{U}$ are marked. Finally, $\mathcal{S}$ is indexed according to the marked elements.

Algorithm 2 implements the ideas above. It indexes $\mathcal{R}$ in Line 1–9 and indexes $\mathcal{S}$ in Line 10–15. When $\mathcal{R}$ is indexed, it is sorted first (Line 1), and then each $r_i \in \mathcal{R}$ is indexed (Line 2–9). For $r_i$, the hash code of $r_i.set$ is obtained by invoking hashAset (see Sect. 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 4–9). If $|r_i.set| = 1$, then item $\langle r_i.ID, - \rangle$ is appended to $I_\mathcal{R}(e_1^{(r_i)})$ (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 $I_\mathcal{R}(e_1^{(r_i)})$ (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 first (Line 10), and then each $s_i \in \mathcal{S}$ is indexed (Line 11–15). The hash code of $s_i.set$ is obtained and 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 14–15).

**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 Algorithm 1 is the expected index structure. The time complexity of Algorithm 1 is postponed to Sect. 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, why does $I_{\mathcal{R}}(e)$ use only two least frequent elements?

The first question can be answered by Lemma 1 and Theorem 1.

**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 Algorithm* 1.

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

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 $e_1, \ldots, e_{i-1}$ is in *r.set*. Similarly, $e_2^{(r)} = e_i$ means that $e_i$ is in *r.set* and only one of $e_1, \ldots, e_{i-1}$ is in *r.set*.

Next, we estimate the probability of $e_i$ being marked by Algorithm 1, i.e., $Pr(e_i \text{ is marked})$. Notice that $e_i$ being marked means it is 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 Algorithm 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.

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

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 $u \ge 2$. □

Now, Lemma 1 helps us obtain non-null inverted lists in $I_{\mathcal{S}}$ and get the theorem below.

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

The second question is answered by the theorem below. It tells us that $\cap_{e \in r.set} I_{\mathcal{S}}(e)$ can be well approximated even if only $k = \frac{\ln |\mathcal{S}|}{\ln |\mathcal{U}| - \ln l_{avg}(\mathcal{S})}$ inverted lists are considered. In fact, $k \ge 3$ only if $|\mathcal{S}| >> |\mathcal{U}|^3$, and $k < 3$ holds for most practical datasets. This positively motivates us to compute the intersection of only two inverted lists of $I_{\mathcal{S}}$ by indexing two least frequent elements in $I_{\mathcal{R}}$. Further improvement in filtering ability is left for hash signatures.

**Theorem 2** *Assume each s.set* ($s \in \mathcal{S}$) *be sampled from* $\mathcal{U}$ *uniformly and independently with average length* $l_{avg}(\mathcal{S})$, *then* $E(|\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)|) = (\frac{l_{avg}(\mathcal{S})}{|\mathcal{U}|})^k \cdot |\mathcal{S}|$ *for* $\forall e_1, \ldots, e_k \in \mathcal{U}$. *Moreover, if* $k = \frac{\ln |\mathcal{S}|}{\ln |\mathcal{U}| - \ln l_{avg}(\mathcal{S})}$, *then* $E(|\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)|) = 1$ *for* $\forall e_1, \ldots, e_k \in \mathcal{U}$.

***Proof*** We first compute the probability of $j$ ($1 \le j \le |\mathcal{S}|$) belonging to $I_{\mathcal{S}}(e)$ for $\forall e \in \mathcal{U}$. After that, we obtain the probability of $j$ belonging to $\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)$ and compute $E(|\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)|)$.

First of all, for $\forall e \in \mathcal{U}$ and $\forall s_j \in \mathcal{S}$ ($1 \le j \le |\mathcal{S}|$), we have

$$
\begin{aligned}
\Pr(j \in I_{\mathcal{S}}(e)) &= \Pr(e \in s_j.set) \\
&= 1 - \Pr(e \notin s_j.set) \\
&= 1 - \binom{|\mathcal{U}|-1}{l_{avg}(\mathcal{S})} \Big/ \binom{|\mathcal{U}|}{l_{avg}(\mathcal{S})} \\
&= l_{avg}(\mathcal{S})/|\mathcal{U}|.
\end{aligned}
$$

Since $s_j.set$ is sampled independently, we know

$$\Pr(j \in \cap_{i=1}^{k} I_{\mathcal{S}}(e_i)) = \Pi_{i=1}^{k} (j \in I_{\mathcal{S}}(e_i)) = l_{avg}^{k}(\mathcal{S})/|\mathcal{U}|^k.$$

Therefore,

$$E(|\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)|) = \sum_{j=1}^{|\mathcal{S}|} 1 \cdot \Pr(j \in \cap_{i=1}^{k} I_{\mathcal{S}}(e_i)) = (l_{avg}(\mathcal{S})/|\mathcal{U}|)^k |\mathcal{S}|.$$

Finally, taking $k$ as variable and solving equation $1 = (l_{avg}(\mathcal{S})/|\mathcal{U}|)^k|\mathcal{S}|$, we know that if $k = \ln|\mathcal{S}|/(\ln|\mathcal{U}| - \ln l_{avg}(\mathcal{S}))$ then $E(|\cap_{i=1}^{k} I_{\mathcal{S}}(e_i)|) = 1$. □

### 3.2 The Join Algorithm

The basic idea of FreshJoin is similar to that of SHJ [11]. That is to use bitwise operations on hash signatures of sets to prune away as many as possible set-pairs whose subset relationships need not to be verified, because bitwise filter is much more economic than the verification. FreshJoin accomplishes this more efficiently by applying appropriately three kinds of signatures indexed in freshIndex. On the one hand, it uses the second type of signatures of tuples in $\mathcal{R}$ (i.e., the least frequent elements) 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 $e$s. 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 $r.set$. On 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 third 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 $I_{\mathcal{R}}(e)$.

Algorithm 2 implements the ideas above. First, it calls freshIndex to build index (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 3–14). 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 7–14). 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 8–10. Otherwise, it computes the intersection (Line 9) and traces the new encountered second 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 each such pair into the bitwise filter (Line 13). Finally, the surviving pairs $\langle r_j, s_k\rangle$ are verified and added into $J$ if $r_j.set \subseteq s_k.set$ (Line 14).

***Example 5*** Consider SCJ with data in Fig. 1 and the index in Fig. 2. $I_{\mathcal{R}}(e_i)$ and $I_{\mathcal{S}}(e_i)$ are joined for each $e_i$. When $e_1$ is considered, since $\langle r_1, e_3\rangle \in I_{\mathcal{R}}(e_1)$, $I_{\mathcal{S}}(e_1) \cap I_{\mathcal{S}}(e_3) = List = \{s_1\}$ is computed. Since $sig_{\mathcal{R}}(r_1)\&\ sig_{\mathcal{S}}(s_1) = sig_{\mathcal{R}}(r_1)$, pair $\langle r_1, s_1\rangle$ is verified but is not output. For $\langle r_2, e_3\rangle \in I_{\mathcal{R}}(e_1)$, since it contains the same $e_3$ as $\langle r_1, e_3\rangle$, 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 pruned away. After all lists are processed similarly, the result shown in Example 1 is obtained.

**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_R, sig_R, I_S, sig_S ← freshIndex (R,S);
2  J ← ∅;
3  for i ← 1 to |U| do
4      If I_R(e_i) = NULL then continue;
5      e←-;
6      List ← I_S(e_i);
7      foreach ⟨j, e_u⟩ ∈ I_R(e_i) do
8          if e ≠ e_u then
9              List ← I_S(e_i) ∩ I_S(e_u);
10             e ← e_u;
11         foreach k ∈ List do
12             If sig_R(r_j)&sig_S(s_k)≠sig_S(r_j) then
13                 continue;
14             If verify(r_j, s_k) then J ← J ∪ {⟨r_j, s_k⟩};
15 return J;
```

*Correctness* We assert that the output $J$ of Algorithm 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, 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 Algorithm 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 Lines 13–15 of Algorithm 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 Lines 7–14 of Algorithm 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, whose proof is postponed to the end of Sect. 4.

**Theorem 3** *Except the costs of verification in Line 14, Algorithm 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 cost to index both input relations, and the latter item is the total cost to perform SCJ. It tells us when FreshJoin performs SCJ efficiently and when inefficiently, which explains well the experimental results in Sect. 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) $O(|\mathcal{R}| \cdot l_{avg}(S))$ time when $|\mathcal{S}| \approx |\mathcal{U}|$; (4)$O(|\mathcal{R}|)$ time when $|\mathcal{S}| \cdot l_{avg}(S) \approx |\mathcal{U}|$; (5) $O(|\mathcal{S}| \cdot l_{avg}(S) \cdot (1 + \frac{|\mathcal{S}| \cdot l_{avg}(S) \cdot w_{sig}}{|\mathcal{U}|^2}))$ 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 the hash signatures for all sets. We first present the basic ideas and the framework of our method, then propose a new hash function, and finally discuss the length $w_{sig}$ with the hash function.

### 4.1 Framework of the Hash Method

The hash method distinguishes three kinds of elements in $\mathcal{U}$, i.e., low (mid 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. We adopt a constant $\alpha$ and consider the accumulated frequencies. Let total $= \sum_{i=1}^{|\mathcal{U}|} f_{\mathcal{S}}(e_i)$. If integers $M$, $H$ satisfy $\sum_{i=1}^{M-1} f_{\mathcal{S}}(e_i) \le \alpha \cdot$ total, $\sum_{i=1}^{M} f_{\mathcal{S}}(e_i) > \alpha \cdot$ total, $\sum_{i=1}^{H-1} f_{\mathcal{S}}(e_i) \le (1-\alpha) \cdot$ total , $\sum_{i=1}^{H} f_{\mathcal{S}}(e_i) > (1-\alpha) \cdot$ total , then all $e_1, \ldots, e_{M-1}$ are called as low-frequency elements, all $e_M, \ldots, e_{H-1}$ are called as mid-frequency elements, and all $e_H, \ldots, e_{|\mathcal{U}|}$ are called as high-frequency elements.

This paper fixes $\alpha = 0.25$ and leaves it for future work to determine $\alpha$ according to the datasets. Thus, if the frequencies follow 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. For example, for datasets in Fig. 1, $e_1$, $e_2$, $e_3$, $e_4$, $e_5$ are low-frequency elements, $e_{10}$, $e_{11}$ are high-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 appear in fewer sets and have sound effects to differ a few sets containing them from many sets not containing them. Similarly, high-frequency elements appear 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.

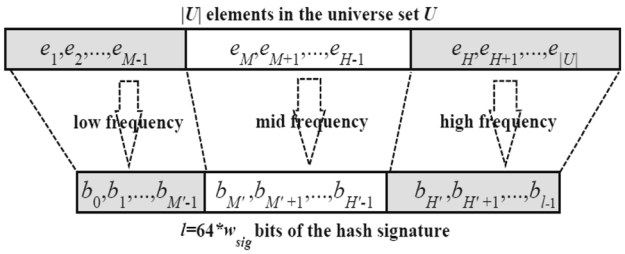



**Fig. 3** Illustration of basic idea in hash method

To do so, all bits in the hash signatures are also partitioned into three parts by two integers $M'$ and $H'$, where $0 < M' < H' < w_{sig} \times 64$. All of $b_0, b_1, \ldots, b_{M'-1}$ are used for low-frequency elements. Similarly, $b_{M'}, b_{M'+1}, \ldots, b_{H'-1}$ are used for mid-frequency elements, and $b_{H'}, b_{H'+1}, \ldots, b_{w_{sig}*64-1}$ are used for high-frequency elements. Each bit in each part is used to represent whether one or more related elements appear in the given set or not. The mapping between elements and corresponding bits is determined by a hash function named freHash (see Sect. 4.2).

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

**Algorithm 3:** hashAset $(aSet)$

**Input:** A set $aSet$ to be hashed, parameters $M$,$M'$, and $H'$
**Output:** The hash signature of the set $aSet$

```
1  initialize all bits of sig to be 0;
2  foreach e_i ∈ aSet do
3      if i < M then                                  // low frequency element
4          j ←freHash (i)%M';
5      else if i < H then                             // mid frequency element
6          j ←M'+freHash (i − M)%(H' − M');
7      else                                           // high frequency element
8          j ←H'+freHash (i − H)%(w_sig * 64 − H');
9      set j-th bit of sig to be 1;
10 return sig;
```

### 4.2 A New Hash Function

Generally speaking, any hash function can map a $e_i$ to a bit $b_j$. For example, $h(i) = i$ or $h(i) = i\%N$ for a suitable $N$, and so on. On the one hand, such functions are helpless in finding a suitable $w_{sig}$ which should be adaptive to input relations. For example, PTSJ [4] just adopted $w_{sig}$= $\min\{\frac{1}{2}l_{avg}(S), |\mathcal{U}|, 256\}$ heuristically. On 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 also distinguishes low (mid and high)-frequency elements. The smaller the $i$ is, the lower frequency the $e_i$ has and fewer such elements should share a common bit. Similarly, the bigger the $i$ is, the higher frequency the $e_i$ has and fewer such elements should share a common bit. Here comes the formal definition of freHash. It is clear that, taking the binary representation of $i$ as input, an algorithm can compute freHash $(i)$ in $O(\log i)$ time.

**Definition 3** 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 $(i)$=$\sum_{k=0}^{\lfloor \log_2 i \rfloor} a_k \cdot k$.

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 $(106) = 1 + 3 + 5 + 6 = 15$.

Figure 4 presents freHash $(i)$ for all $1 \le i \le 128$. Clearly, as expected, freHash roughly implements the ideas above i.e., fewer elements with both low and 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 $1 \le i \le 32$ and lower left part under the blue dashed line is for $1 \le i \le 70$.

It is easy to verify freHash $(i)$= $\sum_{k=0, a_k \in \{0,1\}}^{\lfloor log_2 i \rfloor} a_k \cdot k \le \frac{\lfloor \log_2 i \rfloor (\lfloor \log_2 i \rfloor + 1)}{2}$.

**Definition 4** Let $m_{fh}(n)$ be $\frac{\lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1)}{2}$ for $n > 0$.

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

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

**Definition 5** For a given $n$ and $0 \le \forall i \le m_{fh}(n)$, define $frehash_n^{-1}(i) = |\{j | 0 \le j \le n \text{ and } i = frehash(j)\}|$, and $frehash^{-1}(n) = \max_{i \le m_{fh}(n)} frehash_n^{-1}(i)$.

In fact, $frehash^{-1}(n)$ is the maximum volume of all buckets when frehash is used to map set $[n]$ to set $[m_{fh}(n)]$. Intuitively, the larger the $frehash^{-1}(n)$ is, the more conflicts frehash will cause. Thus, $frehash^{-1}(n)$ characterizes the filtering ability of the signatures. With the help of Fig. 4, it is easy to check the property below. For example, $frehash^{-1}(n) = 5, 7, 10$, and $\frac{1}{4}\log^2 n = 5.25, 9.39, 11.25$ when $n$=32, 70, 128, respectively. This property will be used to explain why we only differ low-, mid-, and high-frequency elements from each other, later.

**Property 2** $frehash^{-1}(n) \approx \frac{1}{4}\log^2 n$.

### 4.3 The Length and Partition of the Hash Signatures

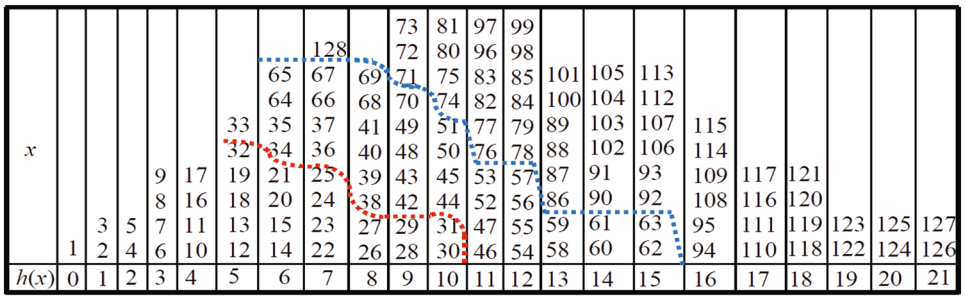
Now, we are ready to discuss the signature length $w_{sig}$ and the values $M', H'$ which partition each hash signature into three parts.

$w_{sig}$ is taken as the minimum of three values (i.e., $w_{sig} = \min\{w_1, w_2, w_3\}$), 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 element if $i < M$ ($M \le i < H$ or $H \le i$, *resp.*). Thus, the input of freHash for low (mid and high)-frequency elements are upper-bounded by $M$, $H - M$ and $|\mathcal{U}| - H + 1$, respectively. According to Property 1, hash signatures for them 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

$$\begin{aligned} \text{len}(M, H, |\mathcal{U}|) &= m_{fh}(M-1) + m_{fh}(H-M) + m_{fh}(|\mathcal{U}| - H + 1) \\ w_1 &= \left\lceil \frac{1}{64} \cdot (\text{len}(M, H, |\mathcal{U}|) + 3) \right\rceil. \end{aligned} \quad (1)$$

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

**Fig. 4** Illustration of freHash




$$w_2 = \left\lceil \frac{1}{64} \cdot (l_{avg}(\mathcal{S}) + 2 \cdot \sigma_S) \right\rceil. \tag{2}$$

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

$$w_3 = \left\lceil \frac{1}{64} \cdot |\mathcal{U}| \right\rceil. \tag{3}$$

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

$$M' = \left\lceil 64 w_{sig} \cdot \frac{m_{fh}(M-1)+1}{\text{len}(M,H,|\mathcal{U}|)+3} \right\rceil \tag{4}$$

$$H' = \left\lceil 64 w_{sig} \cdot \frac{m_{fh}(M-1)+m_{fh}(H-M)+2}{\text{len}(M,H,|\mathcal{U}|)+3} \right\rceil \tag{5}$$

***Remark*** After $M, H, M', H'$ is determined, the mapping from $e_i$ to $b_j$ can be stored in an array, which makes the average running time of hashAset be $O(l_{avg}(\mathcal{S}))$. Therefore, the total time to index both datasets, i.e., the cost of Line 1 of Algorithm 2, is $O(|\mathcal{U}| \log |\mathcal{U}| + (|\mathcal{R}| + |\mathcal{S}|) l_{avg}(\mathcal{S}))$.

Now, we are ready to give the proof of Theorem 3 (see Sect. 3.2).

***Proof*** According to the remark above, the time cost 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 cost for Line 2–Line 13.

First, we analyze the total cost 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 cost of Line 9 is no larger than $\sum_{r \in \mathcal{R}} 2l = 2 \cdot \frac{|\mathcal{R}||\mathcal{S}| l_{avg}(\mathcal{S})}{|\mathcal{U}|}$.

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 $j$s 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 cost of Line 12. First of all, Line 12 needs in worst case $2w_{sig}$ comparisons 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 cost of Line 12 is $\frac{l(l-1)}{|\mathcal{U}|} \cdot 2w_{sig} \cdot |\mathcal{R}|$.

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

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

□

## 4.4 Rationality of Hash Framework

This subsection answers a remaining question, i.e, why set $\mathcal{U}$ only needs to be partitioned into three groups?

To answer this question, we generalize the hash framework in Sect. 4.1 to partition $\mathcal{U}$ into $k$ equi-sized parts and figure out a function to describe the cost of bitwise operation over two hash signatures. Together with Property 2 in Sect. 4.2, we can study the time cost of the improvement in filtering ability, which guides us to choose a rational small $k$.

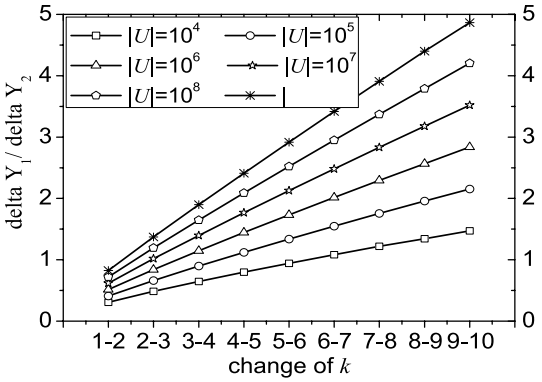



**Fig. 5** Illustration of the change of $\frac{\triangle Y_1}{\triangle Y_2}$

Generally, all epigraphically ordered elements $e_1, \ldots, e_{|\mathcal{U}|}$ in $\mathcal{U}$ can be partitioned into $k$ ($k \geq 1$) groups. The $i$th group consists of $e_{(i-1)\frac{|\mathcal{U}|}{k}+1}, \ldots, e_{i\frac{|\mathcal{U}|}{k}}$. Each group is hashed with frehash independently, and the sequential concatenation of the hash signatures of all groups generates the final signature (as illustrated in Fig. 3).

According to Property 1, the number of bits needed by each group-wise signature is $m_{fh}(\frac{|\mathcal{U}|}{k})$. Totally, $k \cdot m_{fh}(\frac{|\mathcal{U}|}{k})$ bits are needed in each final signature. Thus, $\frac{1}{64}k \cdot m_{fh}(\frac{|\mathcal{U}|}{k})$ 64-bit integers are needed to store each final signature. The cost to perform the bitwise operation between each pair of integers is 2, where one is for loading the operands and the other one is for the operation itself. Therefore, the bitwise filter takes $\frac{1}{32}k \cdot m_{fh}(\frac{|\mathcal{U}|}{k}) \approx \frac{1}{64}k \cdot \log^2(\frac{|\mathcal{U}|}{k})$ time for each pair of final signatures. Thus, we take

$$Y_1(k) = \frac{1}{64}k \cdot \log^2\left(\frac{|\mathcal{U}|}{k}\right)$$

as the cost of bitwise filter when the universe set is partitioned into $k$ equi-sized groups. Notice that both $Y_1'(k) > 0$ and $Y_1''(k) > 0$ hold when $k < |\mathcal{U}|$, which means that the cost of bitwise filter increases more and more rapidly when $k$ increases.

According to Property 2 and the construction of final signature, $\frac{1}{4}\log^2(\frac{|\mathcal{U}|}{k})$ is a good approximation of the maximum volume of all buckets of the final signature. The smaller the value is, the fewer conflicts there are. Thus, we define

$$Y_2(k) = \frac{1}{4}\log^2\left(\frac{|\mathcal{U}|}{k}\right)$$

to represent the filtering ability of our hash method. Notice that $Y_2'(k) < 0$ always holds when $k < |\mathcal{U}|$, which means a larger $k$ promises a stronger filtering ability.

Now, both $Y_1'(k) > 0$ and $Y_2'(k) < 0$ hold, which means that both filtering ability and the cost of bitwise filter increase when $k$ increases. Next, we select $k$ by considering the ratio of their growth speed, i.e., $\triangle_k = |Y_1'(k)/Y_2'(k)|$. Since $\triangle_k$ is an increasing function of $k$, $\triangle_k \leq 1$ means the growth of the cost of bitwise filter is slower than that of filter ability and increasing $k$ is positive. Otherwise, the improvement in filtering ability imposes heavier and heavier burden on the bitwise filter and increasing $k$ is negative. Theoretically, a rational $k$ can be taken as the root of the equation $\triangle_k = 1$. However, this root is difficult to obtain because both $k$ and $\log k$ appear in this equation.

Instead, we try to find an empirical value of $k$. To do so, we set $|\mathcal{U}| = 10^4, \ldots, 10^9$, respectively, and observe the change of $\triangle_k = |\triangle Y_1(k)/\triangle Y_2(k)|$ when increasing $k$ to $k+1$ for $k = 1, \ldots, 9$. Figure 5 presents $\triangle_k$'s changing trend. We find that $\triangle_1 < 1$ always holds, which means partitioning $\mathcal{U}$ into at least two groups is beneficial. Besides, $\triangle_2 \leq 1$ almost always holds, which means that partitioning $\mathcal{U}$ into three groups is beneficial for most datasets from practical applications. On the contrary, $\triangle_k > 1$ holds for most cases when $k \geq 3$, which means partitioning $\mathcal{U}$ into more than three groups is not helpful. As a result, the answer for our question is 3.

## 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.

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

- FreshJoin. Our approach is proposed in Sect. 3.2. where a freshIndex is built up on $\mathcal{R}$ and $\mathcal{S}$ via Algorithm 1 and the hash function proposed in Sect. 4.
- ttjoin. *Prefix-tree-based* (and *signature-based*) algorithm is proposed in [2]. It uses $k$-least frequent elements in each set of $\mathcal{R}$ as the set's signature. We evaluate ttjoin with $k = 1, 2, \ldots, 10$ and choose the smallest running time in each experiment.
- Limit. *Prefix-tree-based* algorithm is proposed in [5], where the depth of the prefix tree is upper-bounded with a parameter $k$. Similarly, we evaluate Limit with $k$=1,2,… ,10 and choose the smallest running time.
- PieJoin. *Prefix-tree-based* algorithm is proposed in [3], where each prefix tree is stored as several arrays.
- Pretti. *Prefix-tree-based* algorithm is proposed in [6].

**Table 2** Characteristic of real datasets

| Dataset | Abbrev. | $\|\mathcal{S}\|$ | $l_{avg}(\mathcal{S})$ | $\|\mathcal{U}\|$ | $M$ | $H$ | $w_{sig}$ | $M'$ | $H'$ |
|---|---|---|---|---|---|---|---|---|---|
| Linux [2] | LINUX | 337,509 | 1.78 | 42,045 | 41,448 | 42,015 | 1 | 43 | 52 |
| Stack [2] | STACK | 545,196 | 2.39 | 96,680 | 81,551 | 95,585 | 1 | 31 | 51 |
| Discogs [2] | DISCO | 7,991,155 | 2.40 | 7,949,791 | 4,682,322 | 7,840,873 | 1 | 26 | 50 |
| Bookcrossing [2] | BOOKC | 337,578 | 3.40 | 105,091 | 98,953 | 104,894 | 1 | 35 | 56 |
| Amazon [2] | AMAZ | 1,231,019 | 4.67 | 2,146,277 | 1,436,024 | 2,133,860 | 1 | 28 | 52 |
| BMS [5] | BMS | 515,597 | 6.53 | 1657 | 1550 | 1650 | 1 | 43 | 56 |
| Kosarak [5] | KOSRK | 990,002 | 8.10 | 41,270 | 39,789 | 41,263 | 1 | 42 | 56 |
| Delicious [2] | DELIC | 666,841 | 11.87 | 685,563 | 647,962 | 685,362 | 1 | 36 | 56 |
| AOL [2] | AOL | 657,427 | 26.09 | 10,154,742 | 4,287,838 | 10,115,374 | 1 | 26 | 52 |
| Twitter [4] | TWIT | 456,626 | 32.53 | 370,341 | 338582 | 369740 | 1 | 34 | 55 |
| LiveJournal [2] | LIVEJ | 3,201,203 | 35.08 | 7,489,296 | 7,456,367 | 7,488,933 | 1 | 41 | 56 |
| OrKut [4] | ORKUT | 3,072,589 | 38.14 | 3,072,626 | 1,962,178 | 2,932,062 | 1 | 25 | 47 |
| Reuters | REUTRS | 283,911 | 213.34 | 781,265 | 404,447 | 706,074 | 2 | 46 | 92 |
| Webbase [4] | WEBBS | 168,704 | 2,976 | 6,142,611 | 5,881,138 | 6,121,663 | 2 | 63 | 101 |
| Enron [2] | ENRON | 516,782 | 111.49 | 435,261 | 430,538 | 435,085 | 3 | 116 | 171 |
| Netflix [5] | NETFX | 17,770 | 5,654 | 480,189 | 340,724 | 460,135 | 4 | 106 | 190 |

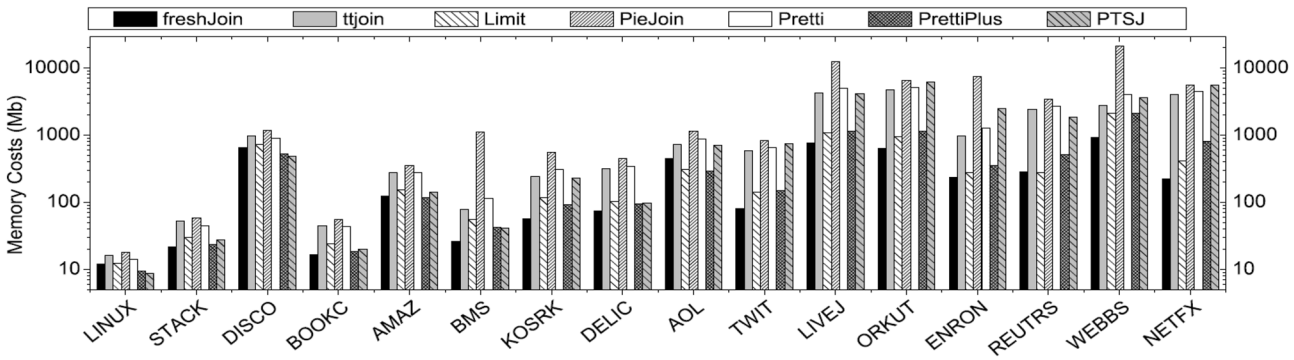



**Fig. 6** The memory cost of different algorithms

- Pretti+. *Prefix-tree-based* algorithm is proposed in [4], where prefix tree is compressed by contracting the non-branching nodes into single nodes.
- PTSJ. *hash-signature-based* algorithm is proposed in [4], where modular is taken as hash function and a PATRICIA Trie is used to help enumerate hash codes.

All these seven 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 [2, 3], the order of elements in sets had a huge impact on the performance of 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 studies, all algorithms were run to do self-join on each dataset.

*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 [2]. Other datasets are different, because we obtain them from KONECT[1] (rather than the addresses given in the studies)

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

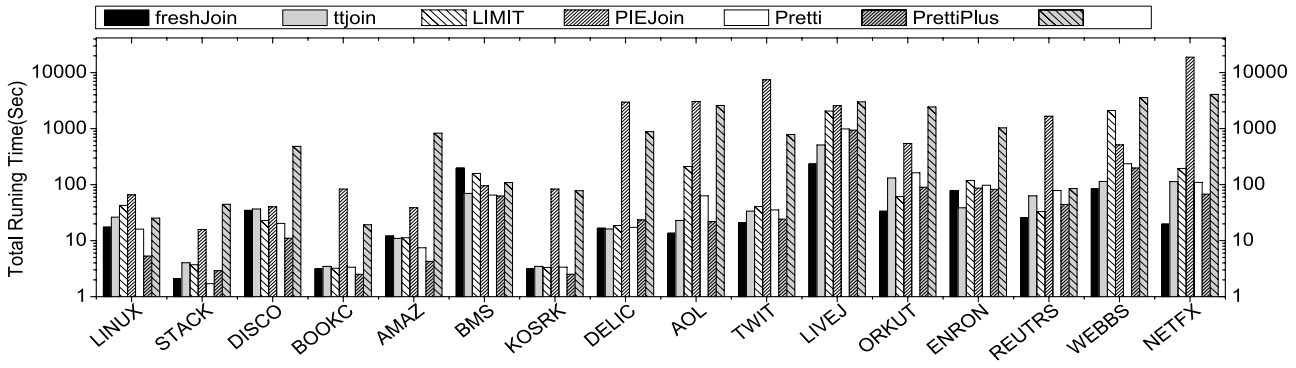



**Fig. 7** The total running time of different algorithms

and extract all sets in each datasets via the tools provided there or the approaches described in studies. 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.

*Exp1: Adaptivity* To evaluate the adaptivity of FreshJoin, we ran FreshJoin on all 16 datasets, and recorded values of $M$, $H$, $M'$, $H'$, and $w_{sig}$, respectively. The results are reported in 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 a Zipfian distribution (see experiments of [2]), then the more skewed the dataset is, the bigger the $M$, $H$ we have; (2) the lengths of hash signatures, which are determined by Formula 4 in Sect. 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 Sect. 4.3, also change adaptively. Therefore, FreshJoin is a parameter-free adaptive algorithm.

*Exp2: Performance* This set of experiments compares FreshJoin with six state-of-the-art algorithms on all 16 datasets. On each dataset, we recorded the maximum space cost of each algorithm and its total running time which includes the time to index the dataset and the time to join the dataset. The results are reported in Figs. 6 and 7, respectively. Besides, for FreshJoin, we also recorded the time to compute the statistics of each dataset and the time to create the index. These results are not reported separately because these costs are very small compared to the total running time. For instance, the time to compute the statistics is no longer than 0.03 seconds and the time to create the index is no longer than 2.5 seconds on each dataset.

For memory usage (see Fig. 6), we find that (1) FreshJoin, Pretti+ and Limit always use less memory, while ttjoin and PieJoin need more memory. This indicates that the prefix trees are space expensive, while both our method and the compressing strategies proposed in [4, 5] are effective. (2) FreshJoin almost always uses least memory, except on Linux and AOL which is the worst cases of FreshJoin according to the remark at the end of Sect. 3.1, but the cost of FreshJoin is still competitive to the cost 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 1. (3) FreshJoin uses much less space than PTSJ. This is because FreshJoin uses a more sucint signature (designed in Sect. 4).

For processing time (see Fig. 7), 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. Besides, the running time of those parametric algorithms is taken as the shortest one among different settings. (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.

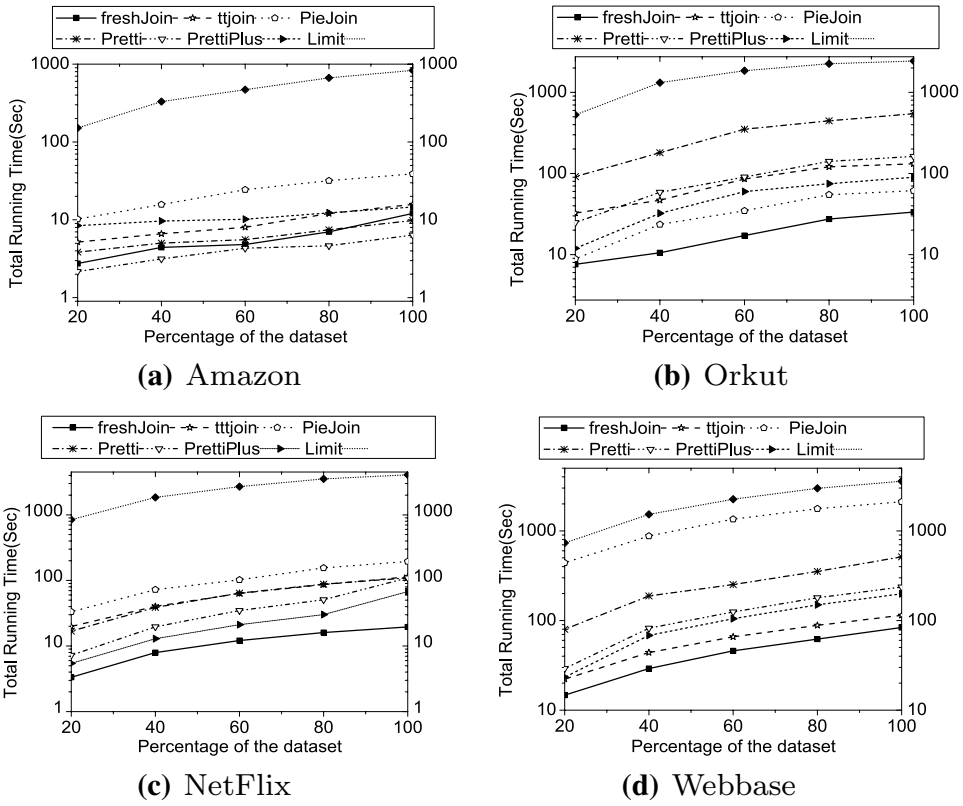



**Fig. 8** Scalability in total running time

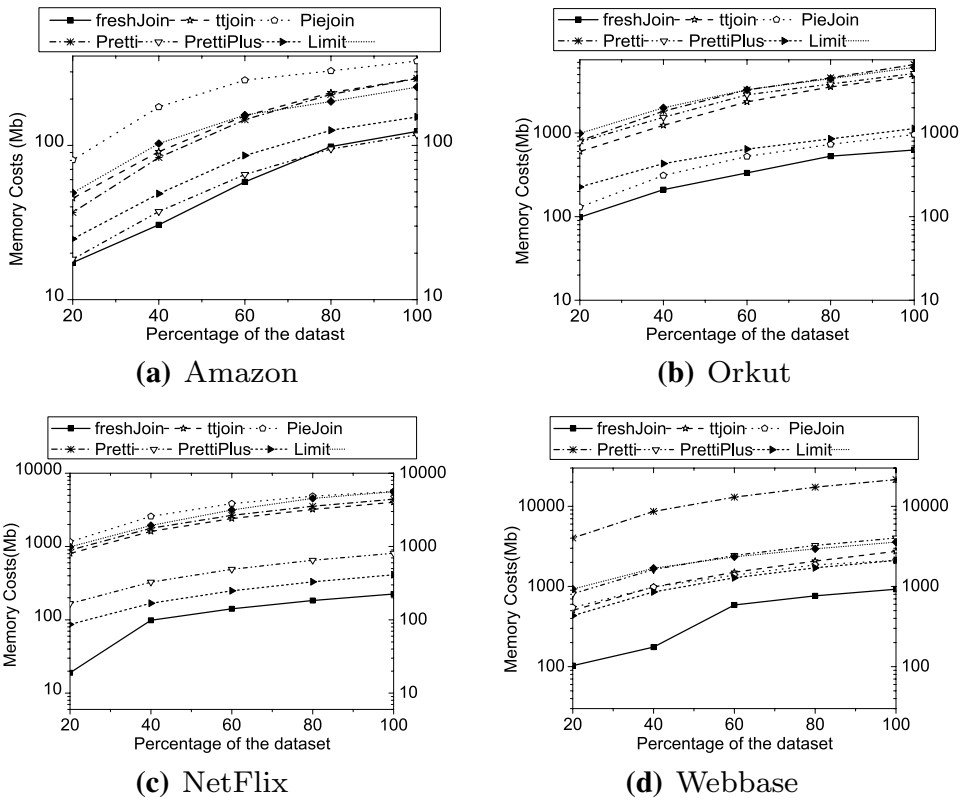



**Fig. 9** Scalability in memory costs

*Exp3: Scalability* The third set of experiments compares the scalability of seven algorithms on four representative datasets. We choose Amazon, OrKut, NetFlix, and Webbase as datasets here, which have different average set lengths $l_{avg}(S)$. Similar in [2], 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 cost of each algorithm are recorded. The results are reported in Figs. 8 and 9, respectively. We find that both the time cost and the space cost of FreshJoin grow 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 in 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 [4]. Therefore, people have studied extensively the theory and engineering of different operations involving set comparison such as containment queries [12, 15–20], similarity joins [13, 21–27], equality joins [7], and containment joins [1–11].

Early work on SCJ mainly focused on disk-based algorithms (e.g., [7, 8, 10, 11]). Although these algorithms have proven quite a effectiveness for joining massive set collections, their performance is bounded by their underlying in-memory processing strategies [4]. For example, PSJ [10] and APSJ [8], two advanced disk-based algorithms, share the same in-memory processing strategy with SHJ [11]. To keep up with ever-increasing data volumes and modern hardware trends, recent work turn to develop next-generation in-memory SCJ algorithms [1–6, 11], which are either signature-based or prefix-tree-based.

All signature-based algorithms (e.g.,SHJ [11], PSJ [10], APSJ [8] and PTSJ [4]) follow the filter-and-refine framework in Sect. 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 signature pairs that may pass through the 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 [4] can be used to alleviate this defect to some extent. In contrast, our method computes signature length adaptively and avoids enumerating signatures by exploiting the least frequent elements to associate signature pairs which are fed to the filter.

Most prefix-tree-based algorithms (e.g., Pretti [6], Pretti+ [4], LIMIT [5], Piejoin [3]) 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 a 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 the tree, many partial results of the intersection can be shared by many tuples in $\mathcal{R}$. Notice that prefix tree is space-costly and traverses of the deep paths are time-costly. So many optimization techniques are adopted. For example, LIMIT [5] limited the height of tree empirically, Pretti+ [4] compressed the prefix tree by merging these non-branching nodes along each path into single nodes, and Piejoin [3] transforms the prefix trees into linear arrays via preorder coding.

The state-of-the-art algorithm ttjoin [2] is 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 a 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. LCJoin [1] proposes to intersect all inverted lists simultaneously with prefix tree to avoid recomputing the intersection and use data partition method to further accelerate the speed of join procedure.

As is pointed out in [2], some existing set similarity search algorithms (e.g., [17, 18, 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 $r \in \mathcal{R}$. And, DivideSkip [18] can also be extended to compute SCJ. Similarly, by 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 SCJ. By setting the error-tolerant threshold as 1 in the index structure for error-tolerant set containment search, the algorithm in [17] can be applied to support SCJ. However, as is shown in the experimental result of ttjoin [2], these renewed algorithms are not as competitive as those SCJ-specific algorithms.

While statistics have been adopted in query optimizations (e.g., [28]), they are not widely used to accelerate SCJ algorithms, to the best of our knowledge. Besides

filter-and-refine frameworks have been widely used in string similarity join (e.g., [29]), they are not widely used in SCJ, except in SHJ [11], PTSJ [4], APSJ [8], and ttjoin [2]).

## 7 Conclusion

This paper revisits the set containment join and proposes a parameter-free join algorithm. It exploits the frequencies of elements to partition the universe set into low-, mid-, and high-frequency elements and maps them separately onto different parts of the hash signatures via a new 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 and efficient.



## References

1. Deng D, Yang C, Shang S, Zhu F, Liu L, Shao L (2019) LCJoin: set containment join via list crosscutting. In: Proceedings of ICDE, pp 362–373
2. Yang J, Zhang W, Yang S, Zhang Y, Lin X (2017) TT-Join: efficient set containment join. In: Proceedings of ICDE, pp 509–520
3. Kunkel A, Rheinländer A, Schiefer C, Helmer S, Bouros P, Leser U (2016) Piejoin: towards parallel set containment joins. In: Baumann P, Manolescu-Goujot I, Trani L (eds) SSDBM, pp 11–22
4. Luo Y, Fletcher G, Hidders J, De Bra P (2015) Efficient and scalable trie-based algorithms for computing set containment relations. In: Gehrke J, Lehner W, Shim K et al (eds) ICDE, pp 303–314
5. Bouros P, Mamoulis N, Ge S, Terrovitis M (2015) Set containment join revisited. Knowl Inf Syst 49:1–28
6. Jampani R, Pudi V (2005) Using prefix-trees for efficiently computing set joins. In: Zhou L, Ooi B, Meng X (eds) DASFAA, LNCS, vol 3453, pp 761–772
7. Mamoulis N (2003) Efficient processing of joins on set-valued attributes. In: Halevy A, Ives Z, Doan A (eds) SIGMOD , pp 157–168
8. Melnik S, Molina H (2003) Adaptive algorithms for set containment joins. ACM Trans Database Syst 28(1):56–99
9. Melnik S, Garcia-Molina H (2002) Divide-and-conquer algorithm for computing set containment joins. In: Jensen C, Jerrery K, Pokorny J, et al (eds) EDBT, LNCS, vol 2287, pp 427–444
10. Ramasamy K, Patel J, Naughton J, Kaushik R (2000) Set containment joins: the good, the bad and the ugly. In: Abbadi A, Brodie M, Chakravarthy S, et al (eds) VLDB, pp 386–395
11. Helmer S, Moerkotte G (1997) Evaluation of main memory join algorithms for joins with set comparison predicates. In: Jarke M, Carey J, Dittrich R, et al (eds) VLDB, pp 386–395
12. Zhu E, Nargesian F, Pu K, Miller R (2016) Lsh ensemble: internet scale domain search. Proc VLDB Endow 9(12):1185–1196
13. Mann W, Augsten N, Bouros P (2016) An empirical evaluation of set similarity join techniques. Proc VLDB Endow 9(9):636–647
14. Luo J, Zhang W, Shi S, Gao H, Li J, Zhang T, Zhou Z (2019) FreshJoin: An Efficient and Adaptive Algorithm for Set Containment Join. In: Shao J et al (eds) Proceedings of APWeb-WAIM 2019: web and big data. LNCS, vol 11642. Springer, pp 175–190
15. Hmedeh Z, Kourdounakis H, Christophides V, Mouza C, Scholl M, Travers N (2012) Subscription indexes for web syndication systems. In: Rundensteiner E Markl, V, Manolescu I, et al (eds) EDBT, pp 312–323
16. Terrovitis M, Bouros P, Vassiliadis P, Sellis T, Mamoulis T (2011) Efficient answering of set containment queries for skewed item distributions. In: Ailamaki A, Amer-Yahia S, Patel J et al (eds) EDBT, pp 225–236
17. Agrawal P, Arasu A, Kaushik R (2010) On indexing error-tolerant set containment. In: Elmagarmid A, Agrawal D (eds) SIGMOD, pp 927–938
18. Li C, Lu J, Lu Y (2008) Efficient merging and filtering algorithms for approximate string searches. In: Alonso G, Blakeley J, Chen A (eds) ICDE, pp 257–266
19. Terrovitis M, Passas S, Vassiliadis P, Sellis T (2006) A combination of trie-trees and inverted files for the indexing of set-valued attributes. In: Yu P, Tsotras V, Fox E, Liu B (eds) CIKM, pp 728–737
20. Yan T, García-Molina Z (1994) Index structures for selective dissemination of information under the boolean model. ACM Trans Database Syst 19(2):332–364
21. Deng D, Li G, Wen H, Feng J (2015) An efficient partition based method for exact set similarity joins. Proc VLDB Endow 9(4):360–371
22. Wang J, Li G, Feng J (2012) Can we beat the prefix filtering? an adaptive framework for similarity join and search. In: Candan K, Chen Y, Snodgrass R et al (eds) SIGMOD, pp 85–96
23. Xiao C, Wang W, Lin X, Shang H (2008) Top-k set similarity joins. In: Alonso G, Blakeley J, Chen A (eds) ICDE, pp 916–927
24. Xiao C, Wang W, Lin X, Yu J (2008) Efficient similarity joins for near duplicate detection. In: Huai J, Chen R, Hon H et al (eds) WWW, pp 131–140
25. Bayardo R, Ma Y, Stikant R (2007) Scaling up all pairs similarity search. In: Williamson C, Zurko M, Patel-Schneider P et al (eds) WWW, pp 131–140
26. Chaudhuri S, Ganti V, Kaushik R (2006) A primitive operator for similarity joins in data cleaning. In: Liu L, Reuter A, Whang K et al (eds) ICDE, pp 5–16
27. Arasu A, Ganti V, Kaushik R (2006) Efficient exact set-similarity joins. In: Dayal U, Whang K, Lomet D (eds) VLDB, pp 918–929
28. Luo J, Zhou X, Zhang Y, Shen H, Li J (2007) Selectivity estimation by batch-query based histogram and parametric method. In: Australia database conference, pp 93–102
29. Luo J, Shi S, Wang H, Li J (2017) FrepJoin: an efficient partition-based algorithm for edit similarity join. Front Inf Technol Electron Eng 18(10):1499–1510