Efficient and scalable trie-based algorithms for computing set containment relations

Article · May 2015

DOI: 10.1109/ICDE.2015.7113293

George H. L. Fletcher
Technische Universiteit Eindhoven
64 PUBLICATIONS **527** CITATIONS
SEE PROFILE

Jan Hidders
Vrije Universiteit Brussel
113 PUBLICATIONS **904** CITATIONS
SEE PROFILE

Paul De Bra
Technische Universiteit Eindhoven
223 PUBLICATIONS **5,732** CITATIONS
SEE PROFILE

Efficient and scalable trie-based algorithms for computing set containment relations

Yongming Luo #1, George H. L. Fletcher #2, Jan Hidders *3, Paul De Bra #4

# *Eindhoven University of Technology, The Netherlands*

1 y.luo@tue.nl, 2 g.h.l.fletcher@tue.nl, 4 debra@win.tue.nl

* *Delft University of Technology, The Netherlands*

3 a.j.h.hidders@tudelft.nl

***Abstract*—Computing containment relations between massive collections of sets is a fundamental operation in data management, for example in graph analytics and data mining applications. Motivated by recent hardware trends, in this paper we present two novel solutions for computing set-containment joins over massive sets: the Patricia Trie-based Signature Join (PTSJ) and PRETTI+, a Patricia trie enhanced extension of the state-of-the-art PRETTI join. The compact trie structure not only enables efficient use of main-memory, but also significantly boosts the performance of both approaches. By carefully analyzing the algorithms and conducting extensive experiments with various synthetic and real-world datasets, we show that, in many practical cases, our algorithms are an order of magnitude faster than the state-of-the-art.**

TABLE I: Example of set-containment join. If we perform a set-containment join ($\bowtie_{\supseteq}$) between user profiles and user preferences, we retrieve matching pairs $\{(u_1, p_1), (u_1, p_2), (u_2, p_3)\}$.

(a) user profiles

id	set	signature
u_1	{b, d, f, g}	0111
u_2	{a, c, h}	1011
u_3	{a, c, d}	1011

(b) user preferences

id	set	signature
p_1	{b, d}	0101
p_2	{b, f, g}	0110
p_3	{a, c, h}	1011

I. Introduction

Sets are ubiquitous in data processing and analytics. A fundamental operation on massive collections of sets is computing containment relations. Indeed, bulk comparison of sets finds many practical applications in domains ranging from graph analytical tasks (e.g., [1]–[3]) and query optimization [4] to OLAP (e.g., [5], [6]) and data mining systems [7].

As a simple example, consider an online dating website where each user has an associated profile set listing their characteristics such as hobbies, interests, and so forth. User dating preferences are also indicated by a set of such characteristics. By executing a *set-containment join* of the set of user preferences with the set of user profiles, the dating website can determine all potential dating matches for users, pairing each preference set with all users whose profiles *contain* all desired characteristics. A concrete illustration can be found in Table I.

In this paper we consider efficient and scalable solutions to the following formalization of this common problem. Consider two relations R and S, each having a set-valued attribute set. The set containment join of R and S ($R \bowtie_{\supseteq} S$) is defined as

$$R \bowtie_{\supseteq} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r.set \supseteq s.set\}.$$

State of the art: Due to its fundamental nature, the theory and engineering of set containment joins have been intensively studied (e.g., [8]–[18]). Existing solutions fall into two general categories: *signature-based* and *information-retrieval-based* (IR) methods. Signature-based methods (e.g., [8]–[12]) encode set information into fixed-length bit strings (called *signatures*), and perform a containment check on the signatures as an initial filter followed by a validation of the resulting pairs using actual set comparisons. IR-based methods (e.g., [13]–[16]) build inverted indexes upon sets storing tuple IDs in the inverted lists. A merge join between inverted lists will produce tuples that contain all such set elements. Typically auxiliary indexes are created to accelerate inverted index entry look-ups and joins.

Most of the focus of the state-of-the-art algorithms has been on disk-based algorithms (e.g., [11]–[13], [15], [16]). Though these algorithms have proven quite effective for joining massive set collections, the performance of these solutions is bounded by their underlying in-memory processing strategies, where less work has been done (see Section II). For example, PSJ [11] and APSJ [12], two advanced disk-based algorithms, share the same in-memory processing strategy with main-memory algorithm SHJ [8], which we'll discuss in detail in Section II-A. To keep up with ever-increasing data volumes and modern hardware trends we need to push the performance of set-containment join to the next level. Therefore, it is essential to revisit (and develop new) in-memory set-containment join algorithms. Such algorithms will serve both as an essential component for main memory databases [19] as well as building blocks and inspiration for external memory and other computation models and platforms. This is challenging because existing work has already investigated many possible optimization techniques, such as bitwise operations [8], caching [13], reusing result set [14] and so on.

Contributions: Nonetheless, by carefully analyzing the existing solutions and bringing in new data structures, in this research we propose two novel in-memory set-containment join algorithms that are in many cases an order of magnitude faster than the previous state-of-the-art. In our study, we scale the relations to be joined along three basic dimensions: set cardinality, domain cardinality, and relation size. Here, *set cardinality* is the size of *set* values in the relations; *domain cardinality* is the size of the underlying domain from which set elements are chosen; and *relation size* is the number of tuples in each relation.

The contributions of our study are as follows:

- We propose two novel algorithms for set-containment join. One is for the low set cardinality, high domain cardinality setting (PRETTI+); the other is for the remaining scenarios (PTSJ). Both algorithms make use of the compact Patricia trie data structure.
- Our PTSJ proposal is a signature-based method. Hence, the length of the signature is a critical parameter for the algorithm's performance. Therefore, we perform a detailed analysis on PTSJ for determining the proper signature length. We also detail how PTSJ can (1) be easily extended to answer other set-oriented queries, such as set-similarity joins, and (2) efficiently be adapted to disk-based environment.
- We present the results of an extensive empirical study of our solutions on a variety of massive real-world and synthetic datasets which demonstrate that our algorithms in many cases perform an order of magnitude faster than the previous state-of-the-art and scale well with relation size, set cardinality, and domain cardinality.

The rest of the paper is organized as follows. In the next section, we introduce the state-of-the-art solutions for set-containment join. In Sections III and IV we propose PTSJ and PRETTI+, our two new algorithms. Section V presents the results of our empirical study of all algorithms. We then conclude in Section VI with a discussion of future directions for research.

II. State-of-the-art Algorithms

In this section we describe two efficient in-memory set-containment join algorithms, SHJ and PRETTI. These solutions are representative of the state-of-the-art, and serve as baseline solutions in our later development and experiments. For simplicity we assume in the following that domain values and tuple IDs are represented as integers.

A. Signature Hash Join

We first introduce the definition of signature [8]. A signature of tuple t ($t.sig$) can be seen as an output of some hash function h (i.e., $t.sig = h(t.set)$) such that

$$t_1.set \subseteq t_2.set \Rightarrow h(t_1.set) \sqsubseteq h(t_2.set).$$

Here the containment relation $\sqsubseteq$ between two hash values is defined as $sig_1 \sqsubseteq sig_2 \Leftrightarrow sig_1 \& \neg sig_2 = 0$, where & and $\neg$ denote bitwise AND and NOT operations. We will also refer to the $\sqsubseteq$ relation as "subset" containment when there is no possibility of confusion.

A straightforward implementation of a signature hash function is as follows: assume the signature length $|t.sig|$ is b bits, all initially set to zero. If integer x is in $t.set$ we set the $(x \bmod b)$th higher-order bit of $t.sig$ to 1. The resulting signature is essentially a *compressed* bitmap representation of $t.set$. In the signature column of Table I we show the 4-bit signature for each set in our example relations. Alphabets are mapped to integers starting from 1, in alphabetical order (i.e., 'a' is mapped to 1, 'b' to 2, and so forth). Note that tuples u_2 and u_3 have the same signature, but different set values.

The Signature Hash Join (SHJ) was proposed by Helmer and Moerkotte [8]. SHJ uses the signature structure as a concise representation for sets, and uses signature comparisons as filtering operations before performing real set comparisons. In the spirit of hash join, SHJ works as follows: (1) for each tuple s in S, compute $s.sig$, and insert ($s.sig$, s) into a hash map (*idx*); (2) for each tuple r in R, compute $r.sig$, enumerate all subsets of $r.sig$, examine all tuples with such signatures in the hash map (hence in S), comparing them with r. Pseudo code of this approach can be found in Algorithm 1 and Algorithm 2. Here we split SHJ into two parts: a generalized signature join framework (Algorithm 1) that can be reused for other algorithms; and, an enumeration algorithm used in SHJ (Algorithm 2) that can be replaced with more efficient algorithms (e.g, Algorithms 4 and 5 below).

Algorithm 1: SIGNATURE_JOIN() signature join framework

```
Input: relations $S$ and $R$
Output: pairs of tuple IDs that have the set
        containment relation
1 create index idx          // e.g., in SHJ is a hashmap
2 for each $s \in S$ do
3     insert (s.sig, s) into idx
4 for each $r \in R$ do
5     subset ← Call subset enumeration algorithm
                              // e.g., SHJ_ENUM(r.sig, idx)
6     for each $s \in subset$ do
7         if r.set $\supseteq$ s.set then
8             output (s, r)
```

Algorithm 2: SHJ_ENUM() subset enumeration of SHJ

```
Input: signature, hash_table
Output: tuple IDs that have signature containment
        relation
1 create list
2 for each subset $\sqsubseteq$ signature do     // enumerate all
3     if subset $\in$ hash_table then
4         for each tuple in hash_table[subset] do
5             add tuple to list
6 return list
```

SHJ inspired other algorithms (e.g., PSJ [11] and APSJ [12]). It is one of the most efficient in-memory solutions for computing set-containment join. One drawback of SHJ comes from line 2 of Algorithm 2, where all subsets of a given signature are enumerated and validated in the hash map. Though the authors provide a very efficient procedure (with bitwise operations) to perform this enumeration, such a mechanism cannot scale with respect to signature length, and therefore cannot scale with relation size and set cardinality. Consequently, all algorithms using this mechanism suffer also from the same problem. In Section III, we provide a solution to this problem, with the introduction of an alternative data structure.

B. PRETTI Join

To the best of our knowledge, PRETTI (PREfix Tree based seT joIn) [14] is the most recent and efficient in-memory set-containment join algorithm. In contrast with SHJ, PRETTI operates on the space of set elements instead of on the space of signatures. In particular, PRETTI works as follows: given relations S and R, first build a prefix tree (trie) based on the ordered set elements of tuples in S; then build an inverted file based on set elements of tuples in R. In the same root-to-leaf path of the trie, tuples of the descendants *contain* tuples of the ancestors. Then when traversing the trie from root to leaf, at each node a list of containment tuples can be generated by joining the tuples in the node and in the inverted list. The list is passed down the trie for further refinement. A sketch of the PRETTI join can be found in Algorithm 3. The recursive call operates on each child of the root node and goes down the tree in a depth-first-search manner.

Figure 1 illustrates the trie structure after inserting sets in user preferences from Table I.

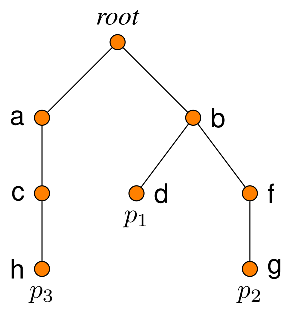


Fig. 1: Trie example for PRETTI, after inserting sets from user preferences (Table I)

Algorithm 3: PRETTI_JOIN() recursively join and output

```
Input: subtree root node, current_list, inverted index idx
Output: pairs of tuple IDs that have signature
        containment relation
// Initially, current_list ← idx[node.label]
1 for each s in node.tuples do
2     for each r in current_list do
3         output (s,r)
4 for each child c of node do
5     child_list = current_list ∩ idx[c.label]
6     PRETTI_JOIN(c, child_list, idx)
```

Assume we have an inverted index created for user profiles from Table I as follows: $\{a{:}\{u_2, u_3\}, b{:}\{u_1\}, c{:}\{u_2, u_3\}, d{:}\{u_1, u_3\}, \ldots\}$. Then when PRETTI executes on the trie in Figure 1, it first finds all tuples that contain element b by probing the inverted index, which is $\{u_1\}$. Then the list is carried to b's children nodes. At node d for instance, the list is joined with the inverted list containing element d, which is $\{u_1, u_3\}$. Since we see one tuple p_1 on the current node, and only u_1 in the list is left, we can conclude that $u_1 \supseteq p_1$. Such actions are performed on all nodes in the trie, and thereby PRETTI finds all containment relations.

PRETTI is a very efficient algorithm. It only traverses the trie once to generate all results. Set comparisons are naturally performed while traversing, and most interestingly, early containment results are reused for further comparisons.

PRETTI has two main weak points. First, many auxiliary data structures such as trie and inverted index are built for the algorithm, which can consume too much space if set cardinality is high. Second, varied-length set comparisons can be time consuming in comparison with fixed-length signature comparisons, especially when set cardinality is high. In our later empirical evaluation we will see that PRETTI can perform quite well for low set cardinality datasets. However, due to excessive main memory consumption and element comparisons, it cannot scale with either larger relations or higher set cardinalities. Later in this paper, we develop extensions to PRETTI to overcome this main-memory consumption problem.

III. Patricia Trie-based Signature Join (PTSJ)

Let's reconsider SHJ from Section II-A. After all signatures are computed, given one signature $r.sig$, SHJ needs two steps to get its subset results: (1) enumerate all subsets of $r.sig$; (2) check whether some subset exists in the hash map entry and perform set comparison afterwards. It is difficult for this mechanism to scale to longer signatures, because the number of possible subsets of a given signature is exponential (2^b) to the signature length b. Therefore in real cases, only part of the signature is used for enumeration purposes (and for creating hash map entries). Based on our experience, this partial signature length cannot even reach 20 bits due to its exponential time complexity. This mechanism essentially limits the possible performance gain of SHJ. However, it is not necessary to enumerate all possible subsets, but rather only those that actually exist in a relation. Hence, we only need $O(|S|)$ time to enumerate subsets of $r.sig$ (that exists in S). This is the core idea of our initial algorithm. We will first introduce our algorithm using a simple binary trie, and later with a Patricia trie.

A. Trie-based Signature Join

Recall that a trie is a basic tree data structure for storing strings. One property of tries is that strings within a subtree share the same path (prefix) from the root to the subtree. Here we use a binary trie, which stores binary strings (i.e., signatures) and tuples associated with a given signature. After we insert all signatures into the trie, since signatures have the same length, we get a trie with the height of signature length. From the root, each level of trie nodes represents one position bit in signatures. Tuple IDs and set values are stored in the leaves of the trie. An example of a binary trie can be found in Figure 2.

When performing a breadth-first search on a trie, in the end we enumerate all existing signatures by visiting the leaves. If we restrict our search at each level of the trie using some given

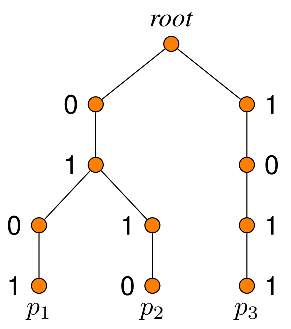


Fig. 2: Trie example, after inserting signatures 0101, 0110, 1011 from user preferences (Table I) into an initially empty trie. Here we let left branches store signatures with prefix bit 0 and right branches store signatures with prefix bit 1.

signature as guidance, we get the subset enumeration algorithm TRIE_ENUM(), given in Algorithm 4. The basic idea is that, while traversing the trie level by level, we are examining all signatures bit by bit. Then if we take the input signature into consideration, the search space shrinks every time a bit "0" is encountered. We use a queue to hold nodes whose prefixes are subsets of the input signature. When Algorithm 4 finishes, all bits of the input signature are examined, and all signatures that are a subset of the input signature are in the queue. We can then directly perform a set comparison of these tuples with the input tuple, by simply plugging in TRIE_ENUM() into line 5 of Algorithm 1.

Algorithm 4: TRIE_ENUM() subset enumeration using trie

```
   Input: signature, trie
   Output: tuple IDs that have signature containment
           relation
1  create queue q
2  i ← 0
3  current_bit ← signature[i++]
4  enqueue trie.root on q
5  while q.top has children do
6      node ← dequeue from q
7      if current_bit = 0 then
           // if node.left exists
8          enqueue node.left on q
9      else
           // if node.left and node.right exist
10         enqueue node.left and node.right on q
11     current_bit ← signature[i++]
12 return q
```

For example, if we want to find containment relations for u_1 in Table I, we first get its signature 0111. Then while we run Algorithm 4, all nodes in the left branch of Figure 2 are visited and placed on the queue. In the end, p_1 and p_2 at leaf nodes are returned.

A limitation of this approach is that there are many unnecessary nodes that only have one child in the trie (which we later refer to as single-branch nodes). We also see this in Figure 2. For k signatures (with b bits each), if there are no single-branch nodes, ideally the trie should have around $2k$ nodes. But instead, it will in the worst case need $k(b - lg_2 k) + 2k$ nodes. The longer the signature, the more single-branch nodes it has. Moreover, these nodes all need to be enqueued and visited. In an empirical study, we witnessed that Algorithm 4 performs slower than SHJ. Therefore we claim that Algorithm 4 is not practical to use, and exclude it from later empirical study (Section V).

B. Introduce Patricia Trie

Knowing what is the weakness, we can improve the design accordingly. To avoid single-branch nodes, we adopt a data structure called Patricia trie [20], [21], which is specifically designed for this purpose. Essentially, a Patricia trie merges single-branch nodes into one node in a trie, so it can guarantee that all nodes have full branches (in our case two-way branches). Of course in the worst case a Patricia trie is not better than a regular trie, but as we'll see in the experiments, that rarely happens for randomly-generated and real-world datasets. Figure 3 shows what a Patricia trie would look like if we insert the same signatures as in Figure 2. First, because there is bit difference on position 0, one node is created on this position. Here, the right branch has no more splitting points, so it directly points to 1011. For the left branch, there is another splitting point on position 2, so another node is created accordingly, and each signature belongs to one of the branches. Overall, 2 extra nodes are created, and there is no single-branch node in the trie.

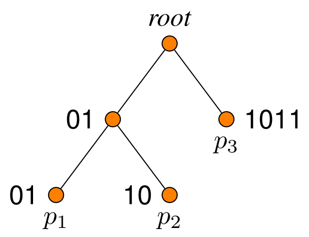


Fig. 3: Patricia trie example, inserting the same signatures as in Figure 2 into a Patricia trie

In this paper we apply a slight modification to the original Patricia trie. In our version of a Patricia trie node, we store (1) pointers to the left and right nodes, (2) the indexes at which point the prefix starts and splits, and (3) the common prefix from the last split point to the current split point.

We define a subset generation procedure on Patricia tries in Algorithm 5. It is similar to Algorithm 4 with the only difference being that, instead of comparing one bit at a time, segments of bits (which come from merged single-branch nodes) are compared at each node. In the end, signatures that have a containment relation are stored in the *result* list instead of queue q. Naturally, we can again reuse Algorithm 1 (by calling PATRICIA_ENUM at line 5) to perform the join. We call this approach **P**atricia **T**rie-based **S**ignature **J**oin (PTSJ).

To continue our example, if we run the same query u_1 (0111) on Figure 3 using Algorithm 5, we still need to visit the left branch of the trie. Only at this time, three instead of six nodes need to be traversed. In practice, signatures can be much longer and sparse (see Section V-B), therefore more node visits are saved compared to Algorithm 4.

Algorithm 5: PATRICIA_ENUM() subset enumeration using Patricia trie

```
Input: signature, patricia_trie
Output: tuple IDs that have signature containment
        relation
1  create queue $q$
2  create list result
3  enqueue patricia_trie.root on $q$
4  while $q \neq \emptyset$ do
5      node $\leftarrow$ dequeue from $q$
6      if node.prefix $\sqsubseteq$ signature.prefix then
7          if node.split = |signature| then
8              add node to result
9          else
10             split_bit $\leftarrow$ signature[node.split]
11             if split_bit = 0 then
12                 enqueue node.left on $q$
13             else
14                 enqueue node.left and node.right on $q$
15 return result
```

C. Cost analysis of PTSJ

In this section we give some cost estimation of PTSJ under simple conditions. Some notation we use are given in Table II. The cost of PTSJ (C_{PTSJ}) can be broken down to

$$C_{\text{PTSJ}} = C_{\text{create\_PT}} + C_{\text{query\_PT}} + C_{\text{compare\_set}},$$

where $C_{\text{create\_PT}}$ is the cost to build the Patricia trie on relation S, $C_{\text{query\_PT}}$ is the cost to compare signatures on the trie, and $C_{\text{compare\_set}}$ is the cost to actually perform set comparisons. We first identify that $C_{\text{create\_PT}}$ and $C_{\text{compare\_set}}$ are not the major cost of PTSJ. Then we dig deeper into $C_{\text{query\_PT}}$, giving an estimation of how many integer comparisons will it cost. We find that under simple natural assumptions, $C_{\text{query\_PT}}$ is mostly influenced by set cardinality c and signature length b. In the end, based on these analyses, we propose a strategy to choose a good signature length for PTSJ.

1) C_{create_PT} and $C_{compare\_set}$:

C_{create_PT}: During Patricia trie creation, at most $2|S|-1$ nodes are created in total. Even in the worst case b nodes are visited during each signature insertion. Obviously, $C_{\text{create\_PT}}$ does not take the major part of PTSJ's running time.

$C_{compare\_set}$: Assume that on average N tuples remain for set comparison for each tuple in R. Then $C_{\text{compare\_set}} = N \times c \times |R|$. It is easy to see that N decreases when signature length grows, and increases when $|R|$ increases. In general this is a small value (from 10's to 100's), proportional to the result output size (see below). Therefore $C_{\text{compare\_set}}$ is also not the major cost of PTSJ.

Estimation of N: To estimate N, we start with a rather simple situation. Consider two signatures d and q, with set cardinalities (and hence number of bits set to 1 in signature) c_d and c_q, resp., and with signature length b. We want to know

TABLE II: Notation for cost analysis

Notation	Explanation
b	Signature length in bits
c	Average set cardinality
d	Domain cardinality, set element domain
Int	Integer size in bits
$\lvert X\rvert$	Size of relation X
H	Average height of Patricia trie
N	Number of tuples in S that have the signature containment relation with some tuple in R
V	Number of trie nodes one query has to visit

what is the probability that $d \sqsubseteq q$. For each element in a set, the probability that it appears on each bit is $\frac{1}{b}$. For $d \sqsubseteq q$ to happen, d should have 1's on only the positions that q has 1's. For each element in d, they have c_q positions to choose from, so each element has the probability $\frac{c_q}{b}$ to be a subset. In total, the probability is $(\frac{c_q}{b})^{c_d}$, and $N = |S| \times (\frac{c_q}{b})^{c_d}$.

We next consider a more complicated scenario. For example, if d's set cardinality is uniformly distributed between 1 and c_d, then the estimated probability of $d \sqsubseteq q$ would be $\frac{p^1+p^2+\ldots+p^{c_d}}{c_d} \approx \frac{p}{c_d \times (1-p)}$, where $p = \frac{c_q}{b}$.

In general, N gets smaller when signature length (b) grows. High set cardinality query (c_q) tends to have more results, while low set cardinality data (c_d) tend to produce more results. All these intuitions are confirmed by our formula. The main take-home message here is that N is a small value, so that set comparisons do not take the significant part of the overall running time.

2) C_{query_PT}:

Let's assume that the number of trie nodes each tuple in R has to visit is V. Then the number of comparisons to be done on the trie is

$$C_{\text{query\_PT}} = |R| \times V \times \left\lceil \frac{b}{H \times \text{Int}} \right\rceil .$$

Here each node on average compares $\frac{b}{|H|}$ bits, which costs $\left\lceil \frac{b}{|H| \times |\text{Int}|} \right\rceil$ actual integer comparisons. We know that $\left\lceil \frac{x}{y} \right\rceil \leq \frac{x}{y} + 1$, so we get the upper bound

$$C_{\text{query\_PT}} \leq |R| \times V \times \left(\frac{b}{H \times \text{Int}} + 1 \right). \qquad (1)$$

We first examine the integer comparisons. For low cardinality settings, signatures are sparse, so two signatures are more likely to share longer prefixes. In the extreme case, all nodes share one path (skewed trie), therefore the average trie height H can be as high as $\frac{b}{2}$. So it is rare for a single node to take more than two integer comparisons. For higher cardinalities, the trie tends to be more balanced, and H is a smaller value closer to $log_2(2|S|)$, but still grows with respect to b. Then we can expect a small but slowly increasing value for comparisons per node. The more important factor however is V.

Estimation of V: There are $\binom{b}{c}$ possible signatures with c bits set to 1. When set cardinality is small (i.e., when $\binom{b}{c} \leq |S|$), it is highly probable that all possible signatures exist in the trie. For example, in the extreme case that set cardinality is 1, there are only $2b$ possible nodes in the trie. Since $2b << 2|S|$, the trie is likely to be full. In such cases, V tends to reach the maximum possible, i.e., $2^c \times H$. Here H is approximately $\frac{b}{2}$.

This becomes less obvious when c and b grow to larger values. In such cases, the trie will not contain all possible cases, and the average height usually does not reach $\frac{b}{2}$. If we have an all-one signature as the query, all nodes ($2|S|$) will be visited. Therefore $2^H = |S|$ (assume balanced trie). If on the lowest level, only one branch is included, the number of nodes to visit becomes $2^{H-1} + 1 \times 2^{H-1}$. Similarly, if single-branches happen for the lowest x levels (which yields the most number of nodes), we get $2^{H-x} + x \times 2^{H-x} = (1+x) \times 2^{H-x}$. Furthermore, if we assume the number of single-branch nodes in a result is proportional to the number of zeros in a signature $(1 - \frac{c}{b})$, so $x = (1 - \frac{c}{b}) \times H$, then, the number of visited nodes is estimated to be

$$V = \left(1 + H \times \left(1 - \frac{c}{b}\right)\right) \times 2^{H \times \frac{c}{b}} \leq (1 + H) \times |S|^{\frac{c}{b}} \quad (2)$$

Here, we see that with the increase of $|S|$, the number of visited nodes increases. Bigger set cardinality also indicates more visited nodes, while longer signatures reduce the number of visited nodes. As we'll see later, we usually select b between $\frac{c}{2} \times Int$ and $c \times Int$, so $(|S|)^{\frac{c}{b}}$ is around 2 even for a million tuples. In such case we say the V is bounded by $O(H)$. And if we bring formula 2 into formula 1, we get $C_{\text{query\_PT}}$ is bounded by $O(c \times |R|)$.

3) Space complexity of PTSJ: Since to build a patricia trie for some relation S, only $2|S|$ nodes are created, and for each tuple the signature size is usually no more than its set values, the space complexity of PTSJ is $O(|S|)$.

D. Choosing the signature length for PTSJ

Because there is no need for exhaustive subset generation, in practice, signature length can be set to thousands of bits in PTSJ without any problem. Generally, longer signatures provide more effective filtering, but bring more signature comparisons and higher main memory consumption. So there is a need for finding the balance point for signature length.

First of all, there is an absolute upper bound for signature length, which is domain cardinality d. Letting $b = d$ essentially makes the signature a bitmap representation of the sets. This number, in many cases can be achieved. For example, for a domain that has 1024 elements, the maximum signature length is $\frac{1024}{Int}$ integers.

It is obvious that there is a lower bound for b as well, which is c. If $b < c$, there is a high chance that all bits in a signature are set to 1, which is not useful anymore.

Apart from these two bounds, we find the "optimum" signature length depends on many properties of input relations, such as set cardinality, domain cardinality, relation size, and data distributions. Among these, we notice both from formula (2) and empirical study (Section V-B) that the set cardinality has a bigger impact on signature length selection, and usually $\frac{c}{2} \times Int \leq b \leq c \times Int$ can yield a good result. This also prevents the algorithm from using more signature comparisons than set comparisons. If not specified otherwise, we use the lower bound of the range ($\frac{c}{2} \times Int$).

Finally, we can set a maximum length in the algorithm, to prevent it from being extremely long. In our experiments, this limit is set to 256 integers.

Overall, our signature length is set to

$$\text{minimum of } \left\{d, \frac{c}{2} \times Int, 256 \times Int\right\}.$$

An empirical validation for this strategy is presented in Section V-B.

E. Extensions to PTSJ

1) Merge identical sets: With the help of the trie, tuples of the same signature are naturally grouped together. If we go one step further, maintaining a mapping list of tuples that have the same set elements, taking them into consideration while output, we save the cost of comparing duplicates over time. This strategy is applied in our PTSJ implementation. It works well without introducing noticeable overhead while creating the trie, and saves quite some comparisons while performing joins, especially for real-world datasets.

2) Superset and set-equality joins: While our algorithms are designed for $R \bowtie_{\supseteq} S$, it can be easily modified to perform $R \bowtie_{\subseteq} S$, in case we want to reuse the existing index on S. Here we take Algorithm 4 as an example to illustrate; Algorithm 5 can be changed in a similar manner. The only place that needs to be touched is the if-else statements (lines 7 to 10). Two case handling statements should be switched, as given in Algorithm 6. Furthermore, in Algorithm 1 the set value containment check (line 7) will change accordingly, to "if $r.set \subseteq s.set$."

Algorithm 6: Replace Algorithm 4 line 7 to 10 for superset join

```
7  if current_bit = 0 then
8      enqueue node.left and node.right on q
9  else
10     enqueue node.left on q
```

Set-equality joins ($R \bowtie_{=} S$) can be answered efficiently as well. In this case, a simple search on the trie will return a list of tuples with the same signature. Further set comparisons are needed to validate the search results. Since we already merge tuples with the same set values, as discussed above in Section III-E1, many set comparisons are saved.

3) Set similarity joins: Apart from being used for set containment computations, a Patricia trie can be (re)used to answer set similarity join [22] queries as well. Set similarity join has been well-studied in the literature [23]. Solutions that make use of a trie have been proposed as well (e.g., [24], [25]), but these do not operate on (and cannot be easily adapted to) the signature space as PTSJ does. For instance, given query signature q, we want to find signatures within hamming distance k. We can use Algorithm 7 to achieve this goal, where we extend Algorithm 4 for illustration purposes. In particular, we use a counter to remember the hamming distance between some prefix and our query. In the end, all signatures (therefore tuples) that are within the distance are in the queue, waiting for other operations (validation, output) to take action. Systems such as OLAP can benefit greatly by reusing one index for different purposes.

Algorithm 7: TRIE_SSJ() hamming distance set similarity join using trie

```
Input: signature, trie, threshold $k$
Output: tuple IDs that have similar signature within
        hamming distance $k$
1  create queue $q$
2  $i \leftarrow 0$
3  current_bit ← signature[i++]
4  enqueue (trie.root, 0) on $q$
5  while q.top has children do
6      (node, i) ← dequeue from $q$
7      if $i \leq k$ then
8          if current_bit = 0 then
9              enqueue (node.left, i) on $q$
10             enqueue (node.right, i+1) on $q$
11         else
12             enqueue (node.left, i+1) on $q$
13             enqueue (node.right, i) on $q$
14         current_bit ← signature[i++]
15 return $q$
```

4) Disk-based algorithm: PTSJ can be easily extended to an external memory setting. A straightforward implementation is to perform a nested-loop join over partitions of the data. Here we partition both relations until one pair of partitions can fit into main memory. Then for each pair of partitions from both relations, we load them into main memory and perform the join. In this case, the algorithm will have a quadratic behavior with respect to the number of partitions. Similar techniques have been applied to other algorithms such as PRETTI. However, as we discussed, PTSJ has a much smaller memory footprint than PRETTI, which makes it more suitable for this strategy. Smarter partitioning techniques (e.g., [11], [12]) can be integrated into PTSJ as well.

F. Discussion

SHJ can be viewed as a one-level, multi-way trie, where each branch starts with a different prefix. PTSJ, on the other hand, is a multi-level, binary trie. The main benefits of PTSJ over SHJ come from longer signatures, which can filter out more unnecessary set comparisons. Furthermore, the trie structure guarantees that only interesting subset prefixes are visited, instead of the whole exponential space.

PRETTI, on the other hand, does make use of a trie structure, but it operates on the set element space instead of signature space. The benefit is that it does not need to be validated twice. The downside, however, is that trie height is as high as the set cardinality, making it only suitable for low set cardinality settings. This brings us to an advanced version of PRETTI, using a Patricia trie.

IV. PRETTI+

Since the Patricia trie is so useful for PTSJ, it is natural to ask if this data structure can be used to advantage with PRETTI. We have integrated a Patricia trie with PRETTI, calling this new join algorithm PRETTI+. Modifications have to be done both on trie construction and on the join procedures.

Inserting sets to the trie can be a bit trickier than with PTSJ, since sets are not necessarily of the same size. In Algorithm 8, we show the trie construction function for PRETTI+. Here we assume each node maintains a prefix, a set of related tuples, and a set of children nodes. The main idea is that, depending on the common prefix between a trie node and the newly arrived set (tuple), the new tuple may be inserted to different positions with respect to the given node. Specifically, the tuple may be inserted to (1) the current root, or (2) some subtree of the current root, or (3) a newly created node that becomes a parent of the current root, or (4) a newly created node that is a sibling of the current root. The core of Algorithm 8 then is to find the correct insertion position.

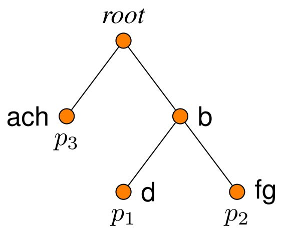


Fig. 4: Trie example for PRETTI+, after inserting sets from user preferences (Table I)

Algorithm 8: PRETTI+_INSERT() trie construction for PRETTI+

```
Input: subtree root node, tuple $s$, cursor on s.set: from
Output: root for the subtree
// insert s.set[from:] to subtree node, here we
treat s.set as a string
1  clen ← |common prefix of node.prefix and s.set[from:]|
2  nlen ← |node.prefix|
3  tlen ← |s.set[from:]|
4  if clen = nlen then
5      if clen < tlen then
6          c ← some child of node that matches
           s.set[(from+clen):]
7          call PRETTI+_INSERT(c, s, from + clen)
8      else                                  // clen = tlen
9          put s into node
10     return node
11 else                                      // clen < nlen
12     if clen = tlen then
13         create new_node for s, insert new_node between
           node and its parent
14     else
15         create new_node as parent for node and tuple
16     return new_node
```

The join operation is almost the same as for PRETTI, except that lists of tuples from the inverted index have to be joined several times in each node, since each node holds several set elements. By replacing a standard trie with a Patricia trie, PRETTI+ consumes much less main memory than PRETTI. However, set comparisons and tuple list joins still take place, same as in PRETTI. As we'll see in our empirical study, PRETTI+ is always a better choice than PRETTI.

V. Empirical Study

In this section we empirically compare the performance of SHJ, PRETTI, PTSJ, and PRETTI+. We first introduce the experiment settings. Then we validate the signature length selection strategy discussed above in Section III-D. After that we conduct the main comparison of the four algorithms on a variety of synthetic and real-world datasets.

A. Experiment setting

1) Synthetic datasets: We create a data generator to generate synthetic relations. The generator can generate relations with varying sizes, set cardinalities, domain cardinalities, and so on. The distribution of data can vary on both set cardinality and elements. The distributions are generated using Apache Commons Math[1], a robust mathematics and statistics package. We start with a simple setting, with uniform distribution on different set cardinalities and set elements. Later we test the algorithms' performance on relations with Zipf and Poisson distributions, which are commonly found in real-world scenarios.

2) Real-world datasets: We experiment with four representative real-world datasets, covering the scenarios of low, medium and high set cardinalities. Some statistics of the datasets[2] are shown in Table III.

TABLE III: Statistics for real-world datasets

data	$\|R\|$	c avg.	c median	d
flickr	3.55×10^6	5.36	4	6.19×10^5
orkut	1.85×10^6	57.16	22	1.53×10^7
twitter	3.7×10^5	65.96	61	1318
webbase	1.69×10^5	462.64	334	1.51×10^7

Flickr-3.5M (flickr): The flickr dataset[3] associates photos with tags [26]. Naturally, here we treat tags as sets, to perform a set-containment join on photo ids. In this way, we create the containment relation between photos. Further operations such as recommendation can be investigated upon such relations. This is a low set-cardinality scenario.

Orkut community (orkut): The Orkut dataset[4] contains relations of people from an online social network and the communities they belong to [27]. Here we treat each person as a tuple and the communities they belong to as a set. Set-containment join in this case, can help people discover new communities and new friends with similar hobbies. Set cardinality for this dataset is higher than Flickr, and we further keep tuples with $c \geq 10$ to exhibit a low-to-medium set cardinality scenario.

Twitter k-bisimulation (twitter): We derive this dataset from paper [28]. Bisimulation is a method to partition the nodes in a graph, based on the neighborhood information of nodes. In this dataset, tuples are the partitions of the graph, and sets are the encoded neighborhood information each partition represents. Here we define the neighborhood of each node to be within 5 steps from the node. On such dataset set-containment join could be used for graph similarity detection and graph query answering. For this dataset, we select tuples with $c \geq 30$, to exhibit a medium set-cardinality scenario.

WebBase Outlinks-200 (webbase): This dataset is a web graph from the Stanford WebBase project [29]. We extract the data[5] using tools from the WebGraph project [30]. We only keep pages that have more than 200 outlinks, following Melnik et al. [12], to exhibit a high set-cardinality scenario.

3) Implementation details: We implement all algorithms in Java. The signature length of SHJ is set to optimal according to paper [8]. The signature length of PTSJ is set as suggested in section III-D. For PRETTI and PRETTI+, we maintain a hash map in each trie node to enable fast access to children while traversing. This is costly but necessary for the algorithm to reach its best performance. Note that here we tried various efficient implementations of hash map (e.g., Fastutil[6], CompactCollections[7], Trove[8]), and we find the HashMap implementation from JDK 7 itself has both the best performance and lowest main memory consumption. The open-source code of all implemented algorithms is available online[9].

4) Test environment: All experiments are executed on a single machine (Intel Xeon 2.27 GHz processor, 12GB main memory, Fedora 14 64-bit Linux, JDK 7). The JVM maximum heap size is set to 5GB, which we think is a decent setting even for today's computers. In the experiments we run each algorithm ten times, and record the average, standard deviation and median of running times. We observe in our measurements that the average gives a good estimate of the running time, and the standard deviation is not significant when compared with the overall time. Hence in the following we only show the average running time. We tend to test with bigger relations when possible, since larger relations and longer running times eliminates the random behavior introduced by OS scheduling. We run programs with `taskset` command, to restrict the execution on one CPU core. The running time we later present include the time to build indexes (e.g., hash map for SHJ and trie structures for the rest algorithms). We notice there is a trend that with the increase of set cardinality, the percentage of index build time over running time decreases. This is due to the fact that bigger set cardinality leads to more set element comparisons, which takes a larger portion of running time accordingly. But in general, the index build time of SHJ and PTSJ are less than 1% and 5% of the overall running time; PRETTI and PRETTI+ on the other hand take more than 70% and 20% of the running time to build indexes.

[1] http://commons.apache.org/proper/commons-math/

[2] Can be downloaded at http://goo.gl/EBaHbA

[3] http://staff.science.uva.nl/~xirong/index.php?n=DataSet.Flickr3m

[4] http://snap.stanford.edu/data/com-Orkut.html

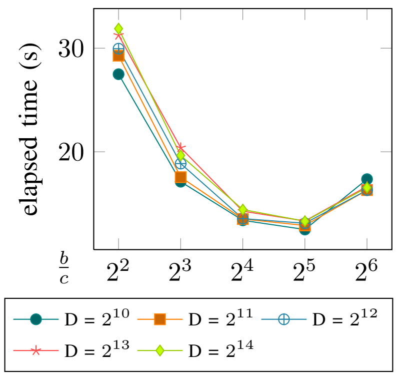
[5] http://law.di.unimi.it/datasets.php

[6] http://fastutil.di.unimi.it/

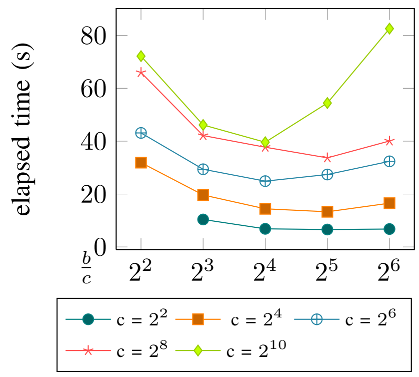
[7] https://github.com/gratianlup/CompactCollections

[8] http://trove.starlight-systems.com/

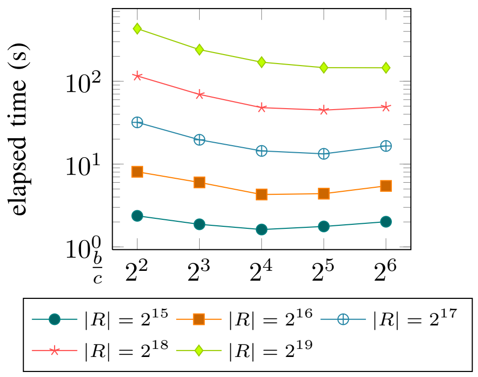
[9] https://github.com/lgylym/scj



(a) Impact of domain cardinality setting



(b) Impact of set cardinality setting



(c) Impact of relation size

Fig. 5: Performance of PTSJ with different signature length settings

For PRETTI and PRETTI+ certain datasets are too big to run in the given memory. In such cases we switch the algorithms to the nested-loop on-disk versions. We notice that PRETTI and PRETTI+ may gain some efficiency by this approach, since the in-memory trie of a partition can be shallower than the global trie. This is more noticeable for high set cardinality scenarios. Overall when switch to disk-based versions, the differences in behavior of PRETTI and PRETTI+ are insignificant, since the algorithms' running times are dominated by computations instead of disk I/Os.

B. *The optimal signature length of PTSJ*

As we discussed, the signature length has a huge impact on PTSJ's performance, sometimes an order of magnitude difference. In Section III-D, we gave some suggestions on how to choose signature length. In this section, we want to empirically validate these suggestions.

Given a dataset, there are three main properties: the relation size, the set cardinality, and the domain cardinality. We want to know how these properties affect the behavior of PTSJ. The strategy of this investigation is to change one property while keeping the other two fixed. By examining the performance under different signature lengths, we can then clearly see whether there is a correlation between a certain property and signature length. Table IV summarizes the settings for this investigation.

TABLE IV: Dataset configurations

fixed parameters	changing parameter
$\|R\| = 2^{17}$, $c = 2^4$	$d \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$
$\|R\| = 2^{17}$, $d = 2^{14}$	$c \in \{2^2, 2^4, 2^6, 2^8, 2^{10}\}$
$c = 2^4$, $d = 2^{14}$	$\|R\| \in \{2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}\}$

Figure 5 shows the performance results of PTSJ, where the x-axis is the ratio between signature length b and set cardinality c. The strategy given in Section III-D suggests that a ratio between 16 and 32 is sufficient. In Figure 5a, we see that indeed, a ratio between 16 and 32 gives the best performance. Domain cardinality does not have a big impact on the signature selection. In Figure 5b we show how the algorithm performs under different set cardinality settings. Again PTSJ finds its best performance point between 16 and 32. We notice that for some high cardinality settings ($c = 2^8, 2^{10}$), comparing signatures themselves becomes an expensive operation. In these cases shorter signatures are preferred in general. Figure 5c shows the impact of relation size over signature length selection. We see a slow trend that when relations grow in size, the optimal signature length tends to move to larger values. This is indicated by formula 2, where $|R|$ is part of the factor. But as we observe, a ratio between 16 and 32 can already give a good result.

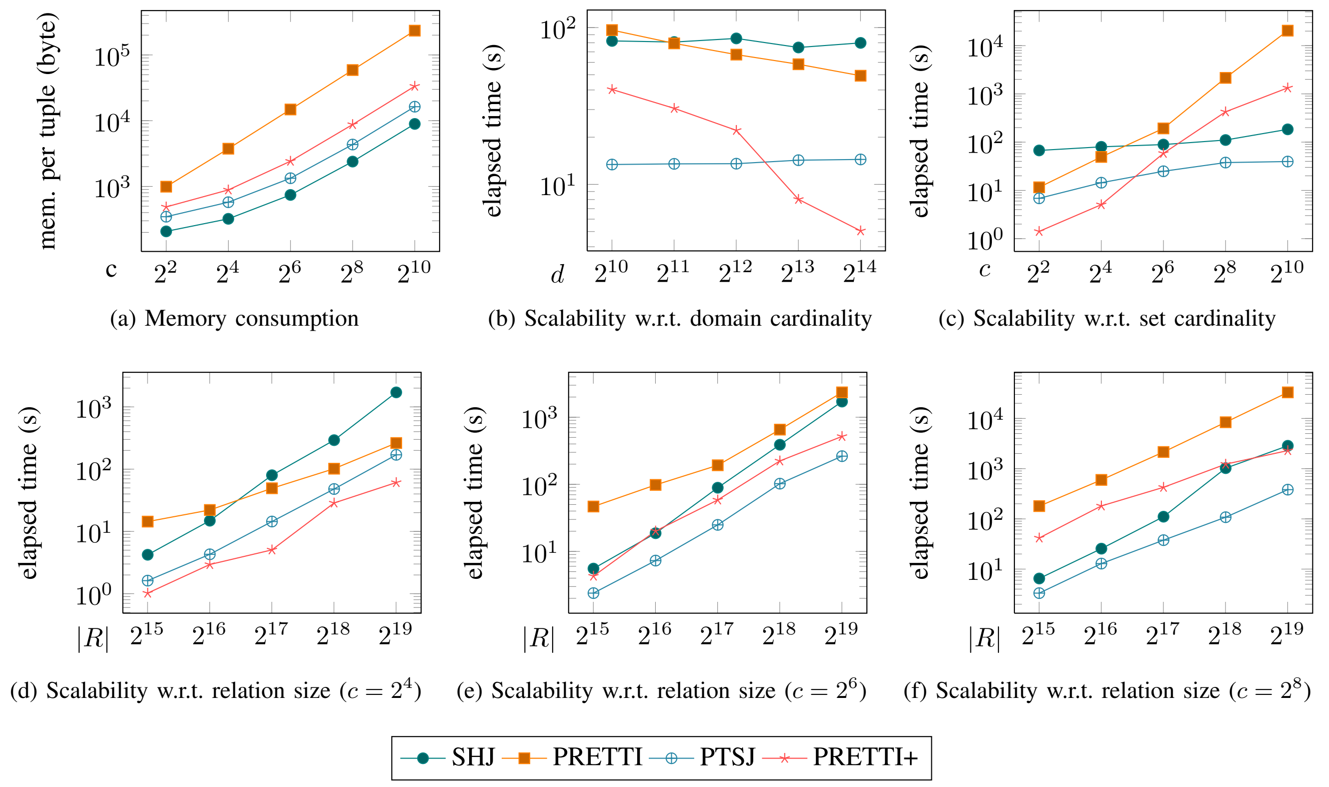
Overall, these experiments support our signature selection strategy of Section III-D. A signature of length between $16c$ and $32c$ is usually a good selection.

C. *Comparison of algorithms*

In this section we discuss the experimental results of the four algorithms on various synthetic datasets. We test on different settings to show the scalability of all algorithms. Figure 6 shows experiments on uniformly distributed datasets. Figure 7 further shows performance on Poisson and Zipf distributions. Dataset configuration is the same as in Table IV.

1) Space efficiency for different algorithms: Main-memory consumption is an essential factor for evaluating main memory algorithms. Low main-memory consumption indicates better scalability of the algorithm with respect to larger datasets. It is not difficult to get a rough estimation of memory consumption for the algorithms mentioned in this paper. The main differences come from the different data structures (indexes) each algorithm uses. For instance, for SHJ, a hash table has to be built; for PRETTI and PRETTI+, a prefix tree and an inverted index; for PTSJ, a patricia trie.

In general, two factors influence memory consumption: (1) relation size $|R|$ and (2) set cardinality c. The influence of relation size is obvious: the number of hash table entries grows linearly with relation size, and so does the size of the prefix tree and inverted index, and the Patricia trie. Set cardinality, on the other hand, has a larger impact on PRETTI and PRETTI+, while SHJ and PTSJ are not so sensitive to it.



(a) Memory consumption (b) Scalability w.r.t. domain cardinality (c) Scalability w.r.t. set cardinality

(d) Scalability w.r.t. relation size ($c = 2^4$) (e) Scalability w.r.t. relation size ($c = 2^6$) (f) Scalability w.r.t. relation size ($c = 2^8$)

Fig. 6: Comparison of different algorithms for uniformly distributed data

We can clearly see this via our experiments. In Figure 6a, we plot, for each join algorithm, with different set cardinality settings, main memory consumption per tuple. Here we note that, though the experiment runs with 2^{17} tuples, the result stays the same for much larger relations. This means that we can estimate how much memory we need, given information about relation size and set cardinality.

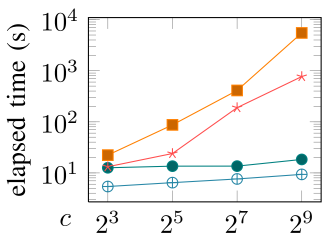
We see that the memory consumption basically has a linear relationship with set cardinality. SHJ, PTSJ and PRETTI+ vary by a constant factor, which is basically the cost of longer signatures (PTSJ), patricia trie (PTSJ and PRETTI+) and inverted index (PRETTI+). PRETTI on the other hand, needs around ten times more main-memory than others. For a relation with set cardinality 2^6, it needs more than 10KB per tuple, which means 10GB for just one million tuples. This empirically substantiates our remarks on PRETTI.

2) Scalability with different domain cardinality settings: Figure 6b depicts performance with different domain cardinality settings. We see that the signature-based solutions (SHJ and PTSJ) are not sensitive to changes in domain cardinality, since they operate on the signature space instead of on the set element space. PRETTI and PRETTI+, on the other hand, operate directly on the set element space. Larger domain cardinality indicates more entries in the inverted index, and shorter inverted lists (therefore faster merge joins on the lists). So PRETTI and PRETTI+ perform better when domain cardinality is high.

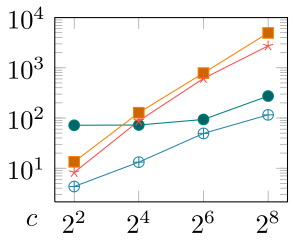
3) Scalability with different set cardinality settings: In order to determine the scalability of the algorithms with respect to set cardinality, we set the relation size to 2^{17}, with average set cardinality varying from 2^2 to 2^{10}. The very high set cardinality scenarios (2^{10}) are not uncommon, especially in the context of graph analytics. We'll see more data of this kind from experiments with real data. In Figure 6c, we see that PRETTI and PRETTI+ are both more sensitive to set cardinalities, compared to the signature-based solutions. When set cardinality is lower (below 2^5), PRETTI+ is a better choice over the other alternatives; but beyond that point, PTSJ is a better choice. In each case, one of our new algorithms will achieve nearly an order of magnitude performance gain over the best of SHJ and PRETTI.

4) Scalability with different relation sizes: Algorithm scalability with respect to relation size may be the most important factor in practice. From Figure 6d to 6f, we show performance with difference set cardinality scenarios ($c = 2^4, 2^6, 2^8$). Just as we saw earlier, for low cardinality settings (Figure 6d), PRETTI+ is a clear winner, followed by PTSJ, PRETTI and SHJ. When set cardinality grows, the advantages of signature-based solutions start to show. PTSJ becomes a better choice over the others. The difference becomes more significant with larger relation sizes. In Figure 6f we see that in many cases in-memory PRETTI (and PRETTI+) cannot finish the experiments, so we switch the algorithm to a disk-based nested-loop version.

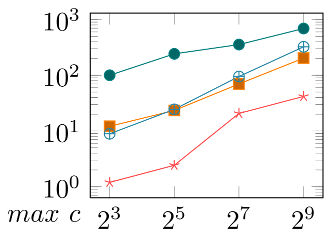
5) Poisson distribution and Zipf distribution: Here we want to determine if different distributions on the set cardinality and set elements have an impact on performance. We test datasets ($|R| = 2^{17}$) with two distributions: Poisson distribution and Zipf distribution, which are widely found in real-world datasets. Distributions are applied to either set cardinality or set



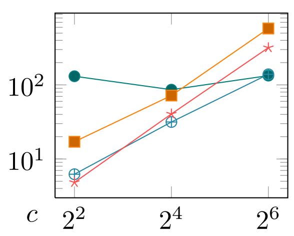
(a) Scalability w.r.t. set cardinality, with poisson distribution on set cardinality



(b) Scalability w.r.t. set cardinality, with poisson distribution on set element



(c) Scalability w.r.t. set cardinality, with zipf distribution on set cardinality



(d) Scalability w.r.t. set cardinality, with zipf distribution on set element

Fig. 7: Comparison of different algorithms for skewed distributions

elements. We expect that the distribution on set cardinality will have a greater impact, as shown previously. Unless specified otherwise, the x-axis shows the average set cardinalities.

In Figure 7a we show datasets with Poisson distribution on set cardinalities. This setting is bad news for PRETTI and PRETTI+, because then the set cardinality can be potentially large. We see that indeed, even when $c = 2^3$, PRETTI and PRETTI+ are not competitive with PTSJ. Indeed, PTSJ performs the best in all cases.

Figure 7b shows Poisson distribution on set elements. This distribution does not make a significant difference for all algorithms, which behave as in Figure 6c.

Zipf distribution on set cardinality favors PRETTI and PRETTI+. As in Figure 7c, we see that PRETTI+ becomes the best solution on all settings. Note that in this case the x-axis is the maximum set cardinality instead of average. Since c follows a Zipf distribution, many sets have small c and only a few have larger ones. In fact, the median set cardinality for the dataset with $max\ c = 2^9$ is only 17. This explains why PRETTI+ performs so well.

Zipf distribution on set elements, as in Figure 7b, does not have a huge impact on performance differences. PRETTI and PRETTI+ perform slightly better than in uniform distribution, since they could produce results earlier due to the nature of Zipf distribution (frequent elements are placed near the trie root).

Overall, our observation is that distributions on set cardinality has a large impact on performance. In such cases, we need to not only examine the average set cardinality, but also the median of set cardinality of data, for choosing the right algorithm. Nonetheless, either PTSJ or PRETTI+ will be the best choice, with sometimes a 10-fold speedup compared with the current state-of-the-art.

D. Experiments on real-world datasets

Figure 8 summarizes performance on various real-world datasets, where we plot the ratio of a certain algorithm's running time over the best algorithm for that dataset. We see that the performance can vary in an order of magnitude for many algorithms. In low-to-medium set cardinality settings (flickr, orkut), PRETTI+ is the clear winner, where signature based methods, even PTSJ, are at least three times slower. SHJ in these two cases runs longer than a day. When it comes to medium-to-high set cardinality settings (twitter), however, the benefit of signatures starts to appear, PTSJ can make the computation 3.6 times faster than the second best (SHJ). For webbase, PTSJ again is at least 8 times faster than the state-of-the-art, 2.6 times faster than PRETTI+.

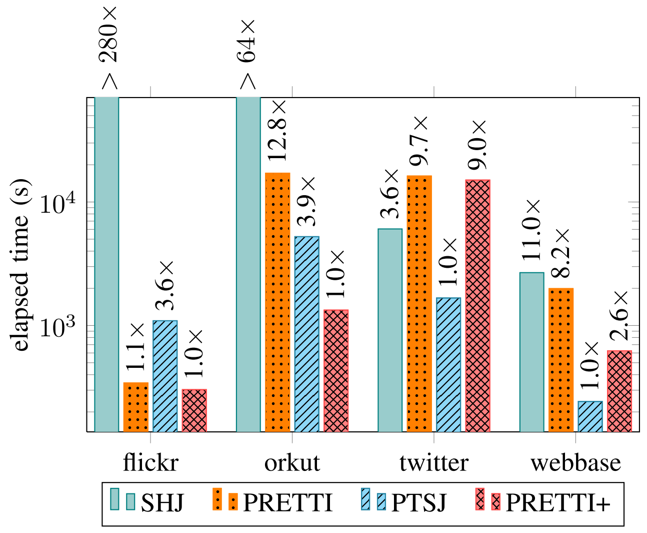


Fig. 8: Algorithm performance comparison for different real-world datasets

VI. Conclusion and Future Work

Motivated by recent hardware trends and practical applications from graph analytics, query processing, OLAP systems, and data mining tasks, in this paper we proposed and studied two efficient and scalable set-containment join algorithms: PTSJ and PRETTI+. The latter is suitable for low set cardinality, high domain cardinality settings, while the former is a more common algorithm suitable for the other scenarios. As shown in the experiments, these two new algorithms can be in many cases remarkably faster than the existing state-of-the-art, and scale gracefully with set cardinality, domain cardinality, and relation size. Detailed analysis has been carried out for PTSJ, especially for finding the optimal value for the critical parameter (signature length). Various extensions of PTSJ make it possible to reuse the index structure to answer other types of join queries, such as set-similarity joins.

Future work: Many interesting topics can be further investigated from this point. Improving the algorithms themselves are a natural first step. For example, more advanced data structures (such as multi-way trie) and join algorithms (such as trie-trie join) certainly need to be considered. Second, extending the algorithms to nontrivial multi-core, external-memory and distributed settings will be essential when relation size goes beyond millions of tuples. Last but not least, it would be interesting to see how our efficient solutions can be adapted to other related data models such as uncertain sets [31] and complex sets [32].

Acknowledgements: The research of YL, GF, JH and PD is supported by the Netherlands Organisation for Scientific Research (NWO). We thank the reviewers for their insightful comments.

References

[1] A. Shkapsky, K. Zeng, and C. Zaniolo, “Graph queries in a next-generation datalog system,” in *VLDB*, Trento, Italy, 2013, pp. 1258–1261.

[2] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren, “A structural approach to indexing triples,” in *ESWC*, Crete, Greece, 2012, pp. 406–421.

[3] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, “gStore: a graph-based SPARQL query engine,” *The VLDB Journal*, vol. 23, no. 4, pp. 565–590, 2014.

[4] B. Cao and A. Badia, “SQL query optimization through nested relational algebra,” *ACM Trans. Database Syst.*, vol. 32, no. 3, pp. 1–46, Aug. 2007.

[5] C. Li, B. He, N. Yan, and M. Safiullah, “Set predicates in SQL: Enabling set-level comparisons for dynamically formed groups,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 2, pp. 438–452, 2014.

[6] A. Badia and A. Wagner, “Complex SQL predicates as quantifiers,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 7, pp. 1617–1630, 2014.

[7] R. Rantzau, “Processing frequent itemset discovery queries by division and set containment join operators,” in *DMKD*, San Diego, California, 2003, pp. 20–27.

[8] S. Helmer and G. Moerkotte, “Evaluation of main memory join algorithms for joins with set comparison join predicates,” in *VLDB*, Athens, Greece, 1997, pp. 386–395.

[9] ——, “A performance study of four index structures for set-valued attributes of low cardinality,” *The VLDB Journal*, vol. 12, no. 3, pp. 244–261, 2003.

[10] S. Helmer, R. Aly, T. Neumann, and G. Moerkotte, “Indexing set-valued attributes with a multi-level extendible hashing scheme,” in *DEXA*, Regensburg, Germany, 2007, pp. 98–108.

[11] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik, “Set containment joins: The good, the bad and the ugly.” in *VLDB*, Cairo, Egypt, 2000, pp. 351–362.

[12] S. Melnik and H. Garcia-Molina, “Adaptive algorithms for set containment joins,” *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 56–99, Mar. 2003.

[13] N. Mamoulis, “Efficient processing of joins on set-valued attributes,” in *SIGMOD*, San Diego, California, USA, 2003, pp. 157–168.

[14] R. Jampani and V. Pudi, “Using prefix-trees for efficiently computing set joins,” in *DASFAA*, Beijing, China, 2005, pp. 761–772.

[15] 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*, Arlington, Virginia, USA, 2006, pp. 728–737.

[16] M. Terrovitis, P. Bouros, P. Vassiliadis, T. Sellis, and N. Mamoulis, “Efficient answering of set containment queries for skewed item distributions,” in *EDBT*, Uppsala, Sweden, 2011, pp. 225–236.

[17] D. Leinders and J. Van den Bussche, “On the complexity of division and set joins in the relational algebra,” *J. Comput. Syst. Sci.*, vol. 73, no. 4, pp. 538–549, 2007.

[18] J.-Y. Cai, V. T. Chakaravarthy, R. Kaushik, and J. F. Naughton, “On the complexity of join predicates,” in *PODS*, Santa Barbara, California, USA, 2001.

[19] D. Lomet and P. Larson, Eds., *IEEE Data Eng. Bull., Special Issue on Main-Memory Database Systems*, vol. 36, no. 2, 2013.

[20] D. R. Morrison, “Patricia–practical algorithm to retrieve information coded in alphanumeric,” *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[21] R. Sedgewick, “Radix Search,” in *Algorithms in Java*. Addison-Wesley Professional, 2003.

[22] A. Arasu, V. Ganti, and R. Kaushik, “Efficient exact set-similarity joins,” in *VLDB*, Seoul, Korea, 2006, pp. 918–929.

[23] S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang, and U. Leser, “State-of-the-art in string similarity search and join,” *SIGMOD Rec.*, vol. 43, no. 1, pp. 64–76, May 2014.

[24] J. Feng, J. Wang, and G. Li, “Trie-join: A trie-based method for efficient string similarity joins,” *The VLDB Journal*, vol. 21, no. 4, pp. 437–461, Aug. 2012.

[25] J. Qin, X. Zhou, W. Wang, and C. Xiao, “Trie-based similarity search and join,” in *Proc. of the Joint EDBT/ICDT 2013 Workshops*, Genoa, Italy, 2013, pp. 392–396.

[26] X. Li, C. G. M. Snoek, and M. Worring, “Unsupervised multi-feature tag relevance learning for social image retrieval,” in *CIVR*, Xi’an, China, July 2010, pp. 10–17.

[27] J. Yang and J. Leskovec, “Defining and evaluating network communities based on ground-truth,” in *ICDM*, Brussels, Belgium, 2012, pp. 745–754.

[28] Y. Luo, G. H. Fletcher, J. Hidders, Y. Wu, and P. De Bra, “External memory k-bisimulation reduction of big graphs,” in *CIKM*, San Francisco, CA, USA, 2013, pp. 919–928.

[29] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, “Webbase: A repository of web pages,” *Computer Networks*, vol. 33, no. 1, pp. 277–293, 2000.

[30] P. Boldi and S. Vigna, “The WebGraph framework I: Compression techniques,” in *WWW*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[31] X. Zhang, K. Chen, L. Shou, G. Chen, Y. Gao, and K.-L. Tan, “Efficient processing of probabilistic set-containment queries on uncertain set-valued data,” *Information Sciences*, vol. 196, pp. 97–117, 2012.

[32] A. Ibrahim and G. H. L. Fletcher, “Efficient processing of containment queries on nested sets,” in *EDBT*, Genoa, Italy, 2013, pp. 227–238.