S3J: A Parallel Semi-Stream Similarity Join

Hao Gao, Muhammad Asif Naeem,
School of Computer and Mathematical Sciences,
Auckland University of Technology
Private Bag 92006, Auckland, New Zealand
alexgao726@gmail.com
mnaeem@aut.ac.nz

Christof Lutteroth, Gerald Weber
Department of Computer Science,
University of Auckland
Private Bag 92019, Auckland, New Zealand
christof@cs.auckland.ac.nz
gerald@cs.auckland.ac.nz

ABSTRACT

Semi-stream join algorithms join a continuous stream with a large disk-based relation. While there are efficient semi-stream equijoins for exact matches in the joined data, there are currently no semi-stream similarity joins for approximate matches. The existing similarity join algorithms work either offline (on datasets that are fully known) or on several streams (using a join window), and are less suitable for applications where continuous, immediate and complete similarity join results are required. To address this gap we propose S3J, the first semi-stream similarity join algorithm. To utilize disk and CPU optimally, S3J combines a disk-intensive queue-based semi-stream join approach with a CPU-intensive similarity matching algorithm. The similarity matching algorithm is based on tries to minimize the memory footprint. Moreover, it supports parallel execution to utilize modern multicore CPUs. We provide a cost model for S3J and evaluate its performance empirically.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

Semi-stream join; similarity search; trie

1. INTRODUCTION

Semi-stream joins appear naturally during stream processing when a stream is interpreted in the context of organizational knowledge. They combine stream data with relatively stable but large master data, often with near real-time performance requirements. There are numerous applications of semi-stream joins, such as real-time data warehousing, sensor networks and social media analysis. In most cases, semi-stream join algorithms perform equijoins, i.e. exact matches between attribute values of stream tuples and disk-based master data records.

DOLAP'15, October 23, 2015, Melbourne, Australia.

ACM 978-1-4503-3785-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2811222.2811226 .

There are some applications where equijoins are too strict and similarity joins are preferable, as data is not always labeled with exact attribute values. For example, this is the case when data is entered manually or recognized by machine learning methods. Hash tags in real-time tweets can be seen as keys in some applications, but they inevitably contain some typographical errors. Searching for similar names and related information in a large database is another kind of application of a similarity join. In ETL process of data warehousing, many documents such as orders and reports are usually generated manually, and therefore attributes such as product names and customer addresses also contain typos. A similarity join is required to join this kind of data. This is particularly useful for data cleaning, to find correct master data for slightly incorrect stream attribute values.

Accordingly, semi-stream similarity join algorithms are a promising field of research. There are other similarity joins, which work either offline or purely on streams (without disk-based master data). For offline algorithms, the datasets that are joined must be fully known – a condition that is not given for a continuous data stream. Furthermore, these algorithms often require all data to be fully loaded into main memory, which may not always be possible. Semi-stream joins are designed to work on a continuous stream and can typically handle very large relations, by keeping them on disk and optimizing disk access.

Also, purely stream-based joins cannot easily satisfy the requirements of a semi-stream join. In order to perform a complete join (i.e. all possible pairs of stream tuples are matched), such algorithms would have to keep all previous stream data in memory indefinitely, as any new stream tuple may join with old data. To avoid this, stream join algorithms are typically confined to a join window, i.e. matches are only performed over recent parts of the streams. By contrast, we expect a semi-stream similarity join to produce all join results between stream and master data, and do so efficiently (near real-time) by considering the characteristics of the stream and the disk-based relation.

To address this gap, we propose S3J, the first semi-stream similarity join algorithm. To utilize disk and CPU optimally, S3J combines a disk-intensive queue-based semi-stream join approach with a CPU-intensive similarity matching algorithm. The similarity matching algorithm is based on an ordered tree data structure – a trie – to minimize the memory footprint, and supports parallel execution to utilize modern multicore CPUs. In particular, we make the following main contributions:

Resource utilization: S3J takes advantage of the com-

putational intensity of similarity joins and the IO intensity of semi-stream joins. Many prominent semi-stream joins leave available resources underutilized. For many of these joins, the disk access is the bottleneck, meaning the processor is mostly idle. We show that in scenarios where a reasonable number of matching errors can be expected (i.e. similarity matches), this idle processor time can be brought to good use by performing a relaxed equijoin.

Low memory consumption: The proposed similarity join is based on tries, which have been found in recent research to be a data structure with very low memory consumption [8]. S3J was specifically designed to match a given service rate with minimal main memory consumption.

Near real-time performance: The time complexity of a similarity join is much larger than that of an equijion, which significantly increases the difficulty of executing the similarity join in near real-time. Although tries have a very low memory footprint, they have a high processing cost [8]. To deal with this, S3J can execute semi-stream similarity joins in parallel, making use of multicore CPUs.

Improved bi-trie algorithm for stream joins: We have improved previous techniques to use two tries [8] and considerably improved performance, for a very modest increase in memory consumption.

Cost model and experiments We provide a theoretical cost model for S3J. Furthermore, we report the results of experiments that validate the cost model and systematically explore the behavior of the algorithm.

Section 2 summarizes related work on semi-stream and similarity joins. Section 3 provides background information about tries and fuzzy search. Section 4 presents S3J in its single-threaded form, with details about its algorithm and cost model. Section 5 describes how S3J can be parallelized and how this affects the cost model. Section 6 provides an experimental evaluation. Section 7 sums up conclusions and points out future work.

2. RELATED WORK

Semi-stream joins have been studied for a while with a focus on equijoins. MESHJOIN [13] is a seminal semi-stream join algorithm. It puts available stream data rather than the whole relation into memory to deal with the problem of very large disk-based relations. It can be seen as a CPU-memory trade-off, which solves the problem of limited memory. It employs a hash table to enable equijoins between stream data with the disk-based relations. Chakraborty et al. [4] propose a partition-based algorithm adjusting the partitions loaded into memory to adapt to the arrival frequency of keys of stream tuples. It exploits the spatio-temporal locality of stream data to reduce disk overhead and therefore reduce delay. HYBRIDJOIN [11] provides a near-real-time algorithm to process intermittent streams using an index on the master data. C-MESHJOIN [12] takes advantage of the distribution of stream tuples to boost the performance in real applications where data is not subject to uniform distributions.

Current research on similarity joins mainly falls into one of two categories: gram-based methods and character-based methods. The former usually adopts a filter-and-verify approach [8]. By building an inverted index associating n-grams and string signatures, pruning methods can be applied to reduce similar candidates in the filtering step. The verification step examines the candidates to generate the final set of similar strings. Based on the constraints of grams, positional grams and length difference of similarity strings, Gravano et al. [6] propose count filtering, position filtering and length filtering.

The process of generating grams is very time-consuming. AllPair [2] uses prefix filtering to alleviate this problem. Edjoin [19] lowers the prefix length bound and uses frequency histograms to further prune the candidates. QChunk [14] proposes new signature schemes based on q-chunks that reduce the the number of generated candidates. PPJoin [20] adopts position filtering and suffix filtering to boost the metrics of Jaccard, Cosine and Dice based algorithms. AdaptJoin [17] provides a framework to improve prefix-filtering based algorithms. VChunk [18] adopts variable-length chunks as signatures to organize the inverted index, while FastSS [3] uses substrings as signatures. PassJoin [10] and PartEnum [1] propose two different pigeon-hole principle based schemes to filter the candidates.

Trie-based similarity joins have been studied to implement character-based algorithms. Wang et al. [16] propose TrieJoin and BiTrieJoin taking advantage of the trie structure to join two big datasets efficiently. Jiang et al. [7] design a partition-based parallel algorithm to boost the performance of a trie-based similarity join by utilizing computation resource. Deng et al. [5] propose MASSJOIN, a MapReduce-based framework to explore repetitive patterns of key-value pairs in data.

Jiang et al. [8] provide a comprehensive evaluation and analysis of different algorithms [2, 19, 14, 20, 17, 18, 3, 10, 1, 16, 7]. For edit distance, it finds FastSS, TrieJoin and BiTrieJoin are efficient for small datasets and low thresholds, e.g., Levenshtein distances ≤ 2; FastSS has a higher performance for datasets with short strings; PassJoin performs better for big datasets. For the metrics of Jaccard, Cosine and Dice, AdaptJoin is preferable for big datasets; PPJoin and AdaptJoin perform better for small datasets with high and low thresholds respectively.

3. BACKGROUND

3.1 The Trie Data Structure

A trie is a tree-based key/value multi-map. The keys are strings (over an arbitrary alphabet A) and stored as a path in the tree, hence the trie uses lexical ordering. Every tree node can represent a key and can be associated with the values of the key. The root node represents the empty string. Every node can have up to one child node for each letter in A. Apart from the root, every node is associated with a letter, and with a path. All the descendants of a node share the common prefix represented by the path from the root to the node.

Tries also facilitate the computation of edit distances between strings. If two strings stored in a trie share a common ancestor node, they share the same prefix. A dynamic programming method can be used to search for similar words in a trie by using this characteristic.

A trie is an asymptotically very compact data structure for storing strings. The height of a trie is one more than the length of the longest key in the trie. If we use, e.g., English words as keys, we can construct a wide and flat trie with a fan-out of 26, which makes the operations of insertion, deletion and search fast. Asymptotically a trie can also reduce memory usage compared to storing strings separately, but

		L	e	v	e	n	s	t	a	i	n
	0	1	2	3	4	5	6	7	8	9	10
L	1	0	1	2	3	4	5	6	7	8	9
e	2	1	0	1	2	3	4	5	6	7	8
v	3	2	1	0	1	2	3	4	5	6	7
e	4	3	2	1	0	1	2	3	4	5	6
n	5	4	3	2	1	0	1	2	3	4	5
s	6	5	4	3	2	1	0	1	2	3	4
h	7	6	5	4	3	2	1	1	2	3	4
t	8	7	6	5	4	3	2	1	2	3	4
e	9	8	7	6	5	4	3	2	2	3	4
i	10	9	8	7	6	5	4	3	3	2	3
n	11	10	9	8	7	6	5	4	4	3	2

Table 1: The computation of Levenshtein distance. In fuzzy search, only the entries in red are needed.

the constant overhead for the size of a trie node has to be taken into account. Our experiments in Section 6 show a compression ratio of about 80% for general English words.

3.2 Fuzzy Search

In this section we will rehash the concept of edit distance and its relationship with tries. We call the search for strings with an edit distance smaller than a given c henceforth a *fuzzy search*. An edit distance measures dissimilarities between two strings in terms of some basic operations, where each operation has a certain cost. The edit distance of two strings is the minimum overall cost of the operations that convert one string to the other. Vladimir Levenshtein defined a commonly used edit distance, the Levenshtein distance [9], with three single-character operations:

1. Character Deletion: $axb \rightarrow ab$
2. Character Insertion: $ab \rightarrow axb$
3. Character Substitution: $axb \rightarrow ayb$ (where $x \neq y$)

The Wagner-Fischer algorithm [15] is a dynamic algorithm that can efficiently compute the edit distance between two strings with a time complexity of $O(mn)$, where m and n are the lengths of the two strings. Table 1 illustrates how this algorithm works by building a table of edit distances for substrings.

The Wagner-Fischer algorithm for computing the Levenshtein distance can be adapted to perform fuzzy search efficiently in a trie. The path for one node from the root in a trie represents a string or a prefix of a string. Let $path_j$ be the character sequence from the root to the node j. If node $j+1$ is a descendant of node j, then $path_j$ is a prefix of $path_{j+1}$ that is one character shorter.

Table 1 shows the usual dynamic programming tableau for the algorithm. The string "Levenshtein" can be a path in a trie, and the string "Levenstain" (494 examples of this misspelling in Google) can be a search key. Then every row corresponds to a node in the path. The simplest dynamic algorithm flood-fills the tableau based on the recursive definition of the Levenshtein distance; the resulting value in the bottom-right corner is the Levenshtein distance between the two strings. Already for the isolated distance between two strings, optimization is possible to bring the time complexity down to $O(d_{max}m)$ where m is the length of the shorter one of the two strings and d_{max} is the maximum edit distance.

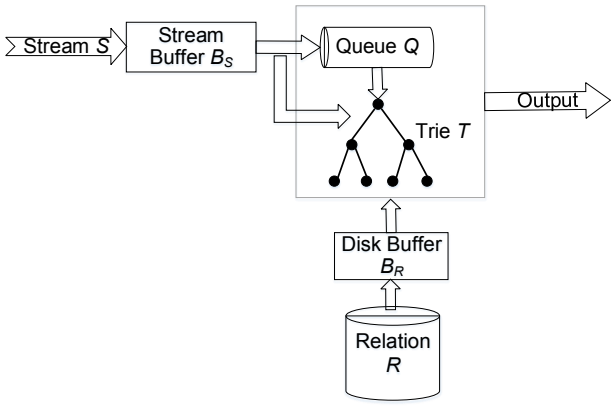

Figure 1: Architecture of the S3J algorithm

The computation of edit distances between a target string and the keys in the trie can be optimized based on the fact that the maximum distance d_{max} is given. In an edit distance table, if an element $d_{i,j} > d_{max}$, it cannot contribute to the computation of the following distances $d_{i+a,j+b}$ where $a \geq 0$ and $b \geq 0$. In Table 1, only the elements in red around the diagonal need to be computed for $d_{max} = 3$. We use this simple improvement in our implementation of fuzzy search. The computation complexity is $O(d_{max}N)$ where N is the node number of the trie and d_{max} is the maximum edit distance.

In this work we restrict the choices of similarity search methods to tries and our improved bi-trie. We do not compare other approaches because we focus here on the parallelization of the similarity search in order to utilize multicore processors. Space consumption apart from the trie is not an issue for the fuzzy search algorithm; the only data structure needed is the table. Thus, there is one array for every node recording edit distance. So the memory usage is HL where H is the trie height and L is the query length.

4. SEMI-STREAM SIMILARITY JOIN (S3J)

We propose S3J, an algorithm that performs a similarity join between stream data and disk relation data. The stream data can be processed in a near real-time fashion, while only a fraction of the disk relation needs to be loaded into memory. S3J creates a balance between computation cost and memory cost. Section 4.1 presents the high-level architecture and design. Section 4.2 describes the details of the algorithm.

S3J, presented as Algorithm 1, can be configured to use any similarity lookup data structure. We investigate here the use of a traditional trie and our improved bi-trie, respectively. Algorithms 1 and 2 represent S3J using a trie, Algorithms 1 and 3 represent S3J using a bi-trie.

4.1 Architecture

The basic components of S3J include the stream buffer B_S, the disk buffer B_R and the trie T as shown in Figure 1. The stream buffer B_S stores the stream tuples waiting to be processed. Because the stream S constantly feeds tuples into the stream buffer, the current fill level of the stream buffer is an important indicator of processing delays in the join. The relation R is assumed to be too large for the main

memory assigned to the algorithm. It is loaded into memory in partitions in a round robin fashion; each round is a table scan.

The algorithm performs the loading of the partitions from the disk in parallel to the similarity join computation. The trie is constructed from the stream tuples, and then relation tuples are used to find stream tuples with similar join attribute in the trie. When a match is found, the corresponding result is output. A stream tuple in the trie stays in the trie until it had a chance to be matched with every relation tuple, i.e. for one table scan of R.

4.2 Algorithm

The procedure of S3J is shown in Algorithm 1. In the algorithm, the relation R is split into k partitions each of which will be loaded into the relation buffer B_R in one iteration. i in the second line points to the partition to load in the coming iteration. Q is a fixed-length queue which records the time order in which a batch of tuples is put into the trie T. From Line 4 to Line 22, a loop is performed to process the infinite stream S. S continuously feeds tuples into the stream buffer B_S.

In every iteration, all the tuples in B_S are put into the trie T, and a copy of the references is put into a container C that is put into the queue Q. The bi-trie is a component for an improved version of the fuzzy search, which will be discussed in Section 4.3 After this, the ith partition of the relation R is loaded into the relation buffer B_R. At the same time, the pointer i is moved to the next partition. As shown in line 12, after every k iterations, i is set to the beginning of the relation. Then, relation tuples are used to traverse the trie, and fuzzy search is performed to find the tuples with similar keys in the trie T.

Finally, the head element C of the queue Q is dequeued. C is a container with references to tuples in the trie T, and all these tuples are removed from the trie. As the length of the queue Q is k, and in k iterations all the relation tuples have traversed the trie, the tuples in the trie T will only be removed after trying to match all the relation tuples. This guarantees that all the stream tuples are fully processed.

4.3 Bi-Trie Fuzzy Search

An efficient similarity join requires a fast fuzzy search algorithm. In this section, we present a modified bi-trie fuzzy search, which improves search performance. Briefly, the bi-trie join employs two trie structures in the Algorithm 1 in Line 7. If two strings $r = r_1r_2...r_m$ and $s = s_1s_2...s_n$ are similar within edit distance d, at least one of the following two conditions must be satisfied:

1. $r_1r_2...r_{\lfloor \frac{n}{2} \rfloor}$ is similar to a prefix of s within edit distance $\lfloor \frac{d}{2} \rfloor$.
2. $r_{\lfloor \frac{n}{2}+1 \rfloor}...r_n$ is similar to a suffix of s within edit distance $\lfloor \frac{d}{2} \rfloor$.

Wang et al. [16] use this property as a filtering condition by building two tries. In this paper, we also build two tries, one normal trie and one trie built from reversed strings. However, we change the fuzzy search algorithm to search on the two tries and directly get results rather than using them as filtering tools. This removes the verification process and therefore boosts the performance.

Algorithm 1 Semi-Stream Similarity Join

```
1: procedure SEMI-STREAM SIMILARITY JOIN
2:     i ← 0
3:     Q ← Queue(length = k)
4:     loop
5:         C ← Container()
6:         for all t ∈ B_S do
7:             add t into trie (or bi-trie) T
8:             add the reference of t into C
9:         end for
10:        enqueue C into Q
11:        load the ith partition of the relation R into B_R
12:        i ← (i + 1) mod k
13:        for all t ∈ B_R do
14:            w ← key(t)
15:            results ← fuzzy_search(w, d_max)
16:            output results
17:        end for
18:        C ← dequeue(Q)
19:        for all t ∈ C do
20:            remove t from T
21:        end for
22:    end loop
23: end procedure
```

We can change the fuzzy search algorithm (as shown in Algorithm 2) on a trie to utilize the two tries. If we want to search similar strings of r within edit distance d in a collection of strings U, we can build a trie T_{left} from the collection U. First, we search $r_1r_2...r_{\lfloor \frac{n}{2} \rfloor}$ with edit distance $\lfloor \frac{d}{2} \rfloor$ in T_{left}. If we cannot reach any node in this step, the search process is finished. Otherwise, we change the edit distance constraints from $\lfloor \frac{d}{2} \rfloor$ to d and continue the search process on the trie. The results of the above process satisfy the first condition of the previous paragraph, and they are similar with r within edit distance d. If we build another trie T_{right} using the reverse form of the strings in the collection U and search the reverse form of the second half of the string $r_{\lfloor \frac{n}{2}+1 \rfloor}...r_n$ on T_{right} following the same procedure, we will get the similar strings satisfying the second condition. However, if a similar string in U satisfies the two conditions, it will exist in both of the results from T_{left} and T_{right}. So a merge process is needed to remove the duplication in the results. The complete procedure is shown in Algorithm 3.

Algorithm 3 is used for S3J with a bi-trie. Firstly, we put every stream tuple t into trie T_{left} and the tuple t with reversed key into trie T_{right}, and then use Algorithm 3 to perform fuzzy search on the two tries. The final results are the combination of the results from the two tries without duplicates.

Figure 2 compares the search speeds of trie and bi-trie. The experiments are performed on fixed tries and bi-tries with 10000 strings from the dictionary dataset described in Section 6.1 and for different maximum edit distances. The experiment corroborates the hypothesis that since the search in the upper levels of the tries is performed with a smaller maximum edit distance, the effort for the search is reduced. It also shows that at least up to distance 5, the cost for odd distances is not noticeably larger than for the next smaller even distance. This effect is theoretically expected, since for odd maximum distances d, the value $\lfloor \frac{d}{2} \rfloor$ is still the same

Algorithm 2 Trie Fuzzy Search

```
1: procedure FUZZY_SEARCH(w, d_max)                ▷ Trie Variant
2:     results ← array()
3:     row ← associative_array()
4:     for all i ∈ length(w) and i ≤ d_max do
5:         row[i] ← i
6:     end for
7:     for all node ∈ sub-nodes of root do
8:         results ← append(results, search(w, node, row,
   d_max, 0))
9:     end for
10:    return results
11: end procedure
12: procedure SEARCH(w, node, last_row, d_max, depth)
13:    results ← array()
14:    row ← associative_array()
15:    for all i ∈ keys of last_row do
16:        d ← edit_distance(w, last_row, depth, node)
17:        if d ≤ d_max then
18:            row[i] ← d
19:            if node contains a key then
20:                results ← append(results, values(node))
21:            end if
22:        end if
23:    end for
24:    if row ≠ ∅ then
25:        for all sub_node ∈ sub-nodes of node do
26:            results ← append(results, search(w,
   sub_node, row, d_max, depth + 1))
27:        end for
28:    end if
29:    return results
30: end procedure
```

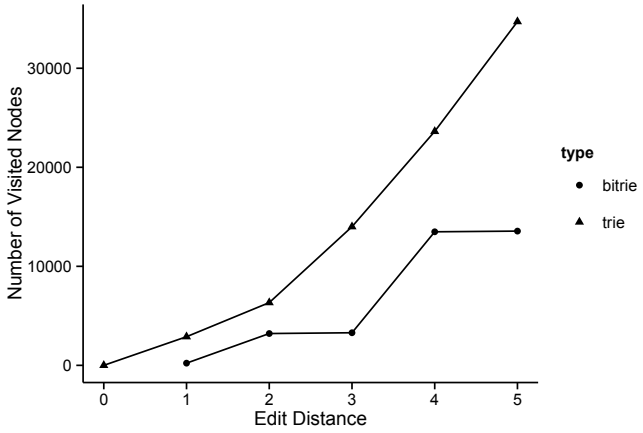

Figure 2: Search speed of trie and bi-trie

as $\lfloor \frac{d-1}{2} \rfloor$. This means that for odd maximum distances the search in the upper levels of the trie requires no more effort than for the next smaller even edit distance. Overall it shows that the bi-trie can reduce the search considerably.

4.4 Cost Model

In this section, we construct the cost model for the algorithms and deduce the optimal values for the key parameters

Algorithm 3 Bi-trie Fuzzy Search

```
1: procedure FUZZY_SEARCH(w, d_max)  ▷ Bi-trie variant
2:     results_left ← bitrie_fuzzy_search(T_left, w, d_max,
   LEFT)
3:     results_right ← bitrie_fuzzy_search(T_right, w, d_max,
   RIGHT)
4:     results ← merge(results_left, results_right)
5:     return results
6: end procedure
7: procedure BITRIE_FUZZY_SEARCH(trie, w, d_max,
   trie_type)                          ▷ By Changing Algorithm 2
8:     add the following lines between Line 1 and Line 2.
9:     "
10:    if trie_type is RIGHT then
11:        w ← reverse(w)
12:    end if
13:    "
14:    replace Line 17 with "if d ≤ adjust(length(w), depth,
   d_max, trie_type)"
15: end procedure
16: procedure ADJUST(length, depth, d_max, type)
17:    if (type = LEFT and depth ≤ ⌊length/2⌋) or (type =
   RIGHT and depth ≤ ⌈length/2⌉) then
18:        return ⌊d_max/2⌋
19:    end if
20:    return d_max
21: end procedure
```

Parameter	Symbol
Size of a stream tuple	v_S
Size of a disk tuple	v_R
Size of the disk relation	R
Partition number	k
Service rate	μ
Ratio of the trie size to node number	α
Cost of one iteration	c_{loop}
Reference size	v_p
Cost of reading one partition of the relation	c_{IO}
Cost of reading one stream tuple	c_S
Cost of inserting one tuple into a trie	c_{insert}
Cost of adding one reference into C	c_{add}
Cost of fuzzy search	c_{search}
Cost of putting an element into a queue	$c_{enqueue}$
Cost of dequeuing an element from a queue	$c_{dequeue}$
Cost of deleting one tuple from a trie	c_{del}

Table 2: Notations of the cost model

to tune the overall performance. The cost model provides a convenient way to evaluate the processing time as well as the memory footprint. The key parameters include the partition number of the disk relation k and the service rate μ. As the algorithm provides guaranteed service, the service rate is equal to the stream tuple arrival rate. The notations for the cost model are listed in Table 2. It is worth to mention that the costs of operations related to the trie are approximate, as they vary with the trie size. Because the trie built in our algorithm tends to be wide and flat, it only slightly influences the real costs in real situations.

Firstly, we construct the model for the trie join, as it is easy to derive the cost model for the bi-trie from the trie. In

the main loop of the algorithm, the time cost of one iteration c_{loop} mainly includes four parts: the time for loading stream data into the trie, the time for reading relation tuples, the time for the similarity join, and the time for deleting fully processed tuples from the trie. The number of processed stream tuples is the product of c_{loop} and the service rate μ, so we can construct Equation 1 to express c_{loop}. Then, we can solve for c_{loop} in Equation 2.

$$\begin{aligned} c_{loop} =& c_{IO} + c_{search}\frac{R}{k} + c_{enqueue} + c_{dequeue} \\ &+ \mu c_{loop}(c_s + c_{add} + c_{insert} + c_{del}) \end{aligned} \tag{1}$$

$$c_{loop} = \frac{c_{IO} + c_{search}\frac{R}{k} + c_{enqueue} + c_{dequeue}}{1 - \mu(c_s + c_{add} + c_{insert} + c_{del})} \tag{2}$$

In Equation 2, $\frac{R}{k}$ is the partition size of the relation data. From Equation 2, we can see c_{loop} increases with decreasing k. To examine the influence of k on the service rate μ, we transform Equation 2 into Equation 3. From Equation 3 we can see that the service rate μ increases with decreasing k. In particular, for $k = 1$ the service rate μ is the largest. It also shows that a bigger trie can improve the service rate.

$$\mu = \frac{1 - \frac{1}{c_{loop}}[c_{IO} + \frac{R}{k}c_{search} + c_{enqueue} + c_{dequeue}]}{c_S + c_{add} + c_{insert} + c_{del}} \tag{3}$$

On the other hand, the processing time increases with decreasing k. For stream tuples, the processing time of a tuple $t_{process}$ is about $c_{loop}k$. From Equation 2 we can deduce $t_{process}$ in Equation 4. As the values of $c_{enqueue}$ and $c_{dequeue}$ are very small compared to c_{IO} and c_{search}, it is safe to omit them. From Equation 4, $t_{process}$ increases with increasing μ. μ increases with decreasing k, so $t_{process}$ increases with decreasing k. This demonstrates that a balance between the processing time and the service rate can be created by adjusting k.

$$t_{process} = \frac{k(c_{enqueue} + c_{dequeue}) + kc_{IO} + Rc_{search}}{1 - \mu(c_S + c_{add} + c_{insert} + c_{del})} \tag{4}$$

The memory is comprised of the stream buffer, the relation buffer, the trie and the queue. In a loop, the number of stream tuples in the stream is $c_{loop}\mu$. The total memory footprint is shown in Equation 5.

$$\begin{aligned} M =& v_S c_{loop}\mu + v_R\frac{R}{k} + \alpha v_p k c_{loop}\mu \\ &+ v_S k c_{loop}\mu \end{aligned} \tag{5}$$

In this equation, the stream buffer size $v_S c_{loop}\mu$ and the relation buffer size $v_R\frac{R}{k}$ increase with decreasing k, while the other two parts, the trie and the queue, decrease with decreasing k. Consequently, the memory consumption can be optimized by varying k. In reality, relation data tends to be very big, and consequently there are limits on how small k can be, in order to reduce the memory usage.

If we substitute Equation 2 into Equation 5, we get the relationship between service rate μ and the memory usage as shown in Equation 6. It shows the memory footprint is proportional to the service rate when $\mu \ll \frac{1}{c_S + c_{add} + c_{insert} + c_{del}}$. When the condition is not satisfied, the memory usage increases exponentially with respect to the service rate μ. However, in Equation 6, μ can be very large as the time complexities of c_S, c_{add}, c_{insert} and c_{del} are $O(1)$. So the memory usage is nearly linearly or quadratically proportional to the service rate for most practical situations.

$$\begin{aligned} M_{trie} =& \frac{\mu(c_{IO} + c_{search}\frac{R}{k} + c_{enqueue} + c_{dequeue})}{1 - \mu(c_S + c_{add} + c_{insert} + c_{del})} \\ &\times (v_S + k\alpha v_p + kv_S) + v_R\frac{R}{k} \end{aligned} \tag{6}$$

For the bi-trie join, the memory needed for the tries is doubled as shown in Equation 7, as there are two tries rather than one. However, as the search speed is faster, the total memory can be smaller for the same service rate by reducing the average stream buffer size and therefore the trie size.

$$\begin{aligned} M_{bi\text{-}trie} =& \frac{\mu(c_{IO} + c_{search}\frac{R}{k} + c_{enqueue} + c_{dequeue})}{1 - \mu(c_S + c_{add} + 2c_{insert} + 2c_{del})} \\ &\times (v_S + 2k\alpha v_p + kv_S) + v_R\frac{R}{k} \end{aligned} \tag{7}$$

5. PARALLELIZATION OF S3J

As we will see in the evaluation section, for larger values of the maximum edit distance, the fuzzy search becomes the dominating factor. However, the fuzzy search can be effciently parallelized in the sense of Amdahl's law, by taking advantage of the constant structure of the trie. Since modern processing hardware provides typically a number of cores, currently typically 4, this can extend the range of input sizes for which the algorithm can be executed at the minimal cost that is incurred by its table scan design.

The only serial fraction (in Amdahl's law terminology) of the fuzzy search phase is the access to the disk buffer and the update of the trie. For the access to the disk buffer there are two obvious alternatives. One alternative is that parallel threads read new tuples from the buffer with exclusive locking. The other alternative is to pre-allocate an equal number of disk buffer tuples to the different threads. In the first alternative one can see intuitively the risk of a serialization between the threads. The second alternative avoids this risk. However, in our experiments the first alternative performs better than the naive version of the second alternative, because in the second alternative not all threads finish at the same time, creating long waiting times. In the experiments, the size of the relation is 547910 tuples. Waiting time is measured with the setting of four parallel threads and an average trie size of 13000. From Table 3 we can see that the maximum extra time cost is $(\text{time}(\textit{synchronized\_method}) + \text{time}(\textit{lock\_unlock})) \times \textit{number\_of\_disk\_tuples} = 11.17736ms$ in the first alternative, while the average waiting time for the second alternative is about $403ms$. Hence the second alternative would need a more complicated strategy which does not look worthwhile given that no problem with mutual delay was observed in the first alternative.

Furthermore, the threads performing the join (the *slave threads*) have to be coordinated with the thread loading the relation (the *master thread*). When the master thread gets a new set of relation tuples, it adds them to the trie and informs the slave threads to execute the join. After every slave thread has finished the join, they inform the master thread to execute the prune operation to remove fully processed stream tuples. The experimental environment is described in Section 6.

Operation	Relative time	time (ms)
Plus (baseline)	1	4e-7
Moving iterator	1	4e-7
Lock-unlock	50	2.162e-5
Fuzzy search	186 435	0.07457429
Waiting time	1 007 500 000	403

Table 3: Time comparison of the main operations

5.1 Efficient Access of Disk Relation

The computation intensity of the algorithm enables efficient loading of disk relation data. Loading disk relation data into the disk buffer can be performed in the background while the program is executing the similarity join between stream and relation tuples. As the similarity join is computation intensive and loading data from disk into memory is IO intensive, both the CPU and IO resources can be effectively utilized. As the relation data can be very large, this can reduce idle IO wait times and increase the overall performance. The extra cost for this design is that the disk buffer is doubled.

The evaluation in Section 6 shows that the time cost of the similarity join is considerable. It is about $10226ms$ for the measurements in Table 3. Based on the type of hard drives and data organization methods, data in the order of Gigabytes can be loaded during the similarity join phase.

5.2 Cost Analysis

In the modified parallel version, the strict serial parts of the algorithm are the cost of reading tuples from the stream and inserting them into a trie. The time complexity of the serial part is $O(1)$ given the trie is wide and flat, which is much smaller than the time complexity of the similarity join operation. So there is only a small part of the algorithm that cannot be parallelized. Based on Amdahl's law, the service rate can be boosted to nearly N times of the original algorithm with efficient IO access, where N is the number of CPU cores.

A cost model can be used to describe the performance increase. The cost of the similarity join c_{join} in a loop is $\frac{R}{kN}c_{search}$. Normally, c_{join} is larger than c_{IO}. By eliminating the cost c_{IO}, the cost model can be expressed in Equations 8 to 11.

$$c_{loop} = \frac{c_{search}\frac{R}{kN} + c_{enqueue} + c_{dequeue}}{1 - \mu(c_s + c_{add} + c_{insert} + c_{del})} \tag{8}$$

$$t_{process} = \frac{k(c_{enqueue} + c_{dequeue}) + \frac{R}{N}c_{search}}{1 - \mu(c_S + c_{add} + c_{insert} + c_{del})} \tag{9}$$

$$\begin{aligned} M_{trie} =& v_S c_{loop}\mu + 2v_R\frac{R}{kN} + \alpha v_p k c_{loop}\mu \\ &+ v_S k c_{loop}\mu \end{aligned} \tag{10}$$

$$\begin{aligned} M_{bi\text{-}trie} =& v_S c_{loop}\mu + 2v_R\frac{R}{kN} + 2\alpha v_p k c_{loop}\mu \\ &+ v_S k c_{loop}\mu \end{aligned} \tag{11}$$

The analysis in Section 4.4 is also applicable. Briefly, the service rate μ increases, and the processing time $t_{process}$ decreases, while the memory increases due to the doubled disk buffer. Especially, since $k(c_{enqueue} + c_{dequeue})$ is very small compared with Rc_{search}, the delay t_{delay} can be reduced to less than one Nth of the original time. For the bi-trie join, the memory allocated for the trie is doubled.

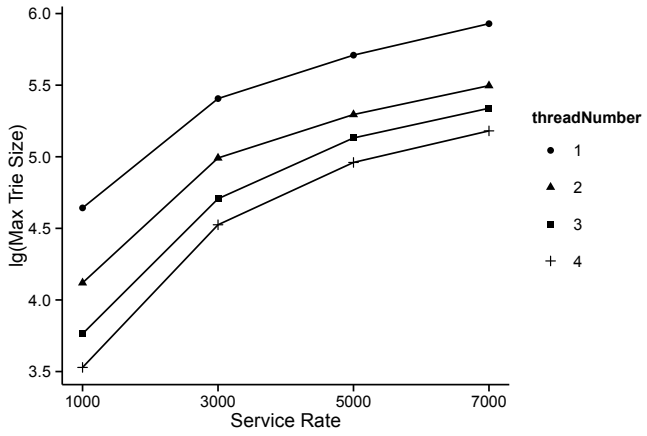

Figure 3: Comparison of different numbers of threads for max. edit distance 1

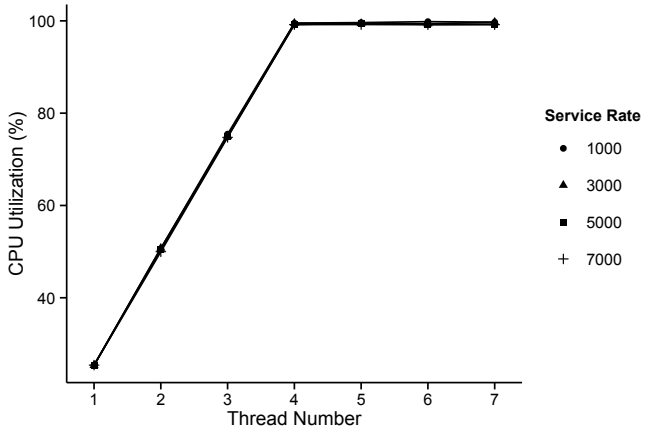

Figure 4: CPU utilization for different numbers of threads

6. EVALUATION

6.1 Experimental Setup

We performed the tests on a PC with Intel Core i5-3570 (3.4G HZ), 8 GB memory and 500 GB HDD, using a 64 bit Windows 7 operating system and a 64 bit Oracle Java JDK 1.8. All tests ran in the server mode of the JVM. The stream and relation data used were a corpus of DBLP authors[1], which had 1,598,437 records with an average length of 14.5, and an English dictionary containing 109,582 words[2]. Typos were randomly injected into the stream tuples. To focus on program performance, the action of finding similar words for

[1] `http://dblp.uni-trier.de/xml/dblp.xml.gz`

[2] `http://www.institute.loni.org/lasigma/ret/products/Burkman/wordsEn.txt`

a word was also performed if the word was already correctly spelled. This made the test results less sensitive to the ratio of errors in the stream.

6.2 Performance Analysis

The results show that the parallel optimization can improve the performance significantly. Figure 3 compares the performance using different numbers of threads for the trie join on the English dictionary. The unit for service rate is tuples/s, and for the trie size it is the number of stream tuples stored in the trie. We can see that for a service rate of about 1000 tuples/s with a maximum edit distance of one, the maximum trie size when using four threads is less than one-tenth of the maximum trie size when using only one thread.

Figure 4 shows the CPU utilization of the trie join. The average CPU usage peaks at nearly 100 percent with four threads as expected with the four core architecture of the hardware that we used. Accordingly, we can not expect any further benefits by increasing the number of threads. We have tested the algorithm up to 7 threads and have observed no benefit in using more than 4 threads, in accordance with the expectations from our hardware setup.

it should be noted that the service rate does not influence the CPU utilization since the algorithm pursues the strategy of serving stream tuples as soon as possible, which means a lower stream tuple arrival speed decreases delay rather than lowering CPU usage.

The ratio of trie size to the number of words in the trie for DBLP Authors is about 9, which varies slightly with trie size. In our current implementation, one node needs 120 Bytes memory on average, so it needs about 1.62 GB memory to store the DBLP Authors in a trie. As the trie takes the majority of the total memory, the trie size can indicate the memory usage. By limiting the trie size to 900000 (about 10000 words, 220 MB memory), the maximum service rate of the bi-trie join is about 10000 tuples/s for edit distance one and 600 tuples/s for edit distances 2 and 3.

Figure 5 shows the relation between the service rate and the maximum trie size on a log-log scale. It reveals the observed asymptotic behavior of the algorithm. The blue line shows the fitting line of maximum trie size and service rate. For the same edit distance, the bi-trie shows a shallower slope and hence a considerably better asymptotic behavior.

The performance statistics for Levenshtein distance 0 are shown for reference – of course its time complexity is much smaller and in this case, the bi-trie cannot improve the performance, so no data for it is shown. Note that the range of service rate shown decreases with higher Levenshtein distances, since these distances are much more costly to process. This is also the reason why Figure 5 (c) shows the range where the bi-trie starts to outperform the trie. In fact, for that edit distance, the simple trie developed stability issues and the algorithm could not be made to deliver higher service rates reliably.

7. CONCLUSION

We have proposed two algorithms, trie join and bi-trie join, to perform semi-stream similarity joins efficiently. Our bi-trie algorithm makes higher maximum edit distances feasible, and odd maximum edit distances can be supported particularly well. A framework was implemented to demonstrate their efficiency and effectiveness. We have also assessed some practical optimizations of the algorithms. The evaluation results show that our algorithms can perform semi-stream similarity joins for short edit distances in a near real-time fashion. Furthermore, we have shown that since similarity joins are compute-intensive, they can make good use of modern multi-core CPUs.

As a future work to improve performance, taking the distribution of stream tuples into account can potentially be better for real-world data. There are many kinds of misspellings that are not subject to uniform distributions. A similarity join algorithm can potentially take advantage of the distribution of such misspellings.

8. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of VLDB*, pages 918–929, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the International World Wide Web Conference*, pages 131–140. ACM, 2007.

[3] T. Bocek, E. Hunt, and B. Stiller. Fast similarity search in large dictionaries. Technical Report ifi-2007.02, University of Zurich, Department of Informatics, 2007.

[4] A. Chakraborty and A. Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *Proceedings of the International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2009.

[5] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 340–351. IEEE, 2014.

[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, and others. Approximate string joins in a database (almost) for free. In *Proceedings of VLDB*, pages 491–500, 2001.

[7] Y. Jiang, D. Deng, J. Wang, G. Li, and J. Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT Workshops*, pages 341–348. ACM, 2013.

[8] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. In *Proceedings of VLDB*, pages 625–636, 2014.

[9] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[10] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. In *Proceedings of VLDB*, pages 253–264, 2011.

[11] M. A. Naeem, G. Dobbie, and G. Weber. X-HYBRIDJOIN for near-real-time data warehousing. In *Advances in Databases, LNCS 7051*, pages 33–47. Springer, 2011.

[12] M. A. Naeem, G. Weber, G. Dobbie, and C. Lutteroth. A generic front-stage for semi-stream processing. In *Proceedings of CIKM*, pages 769–774. ACM, 2013.

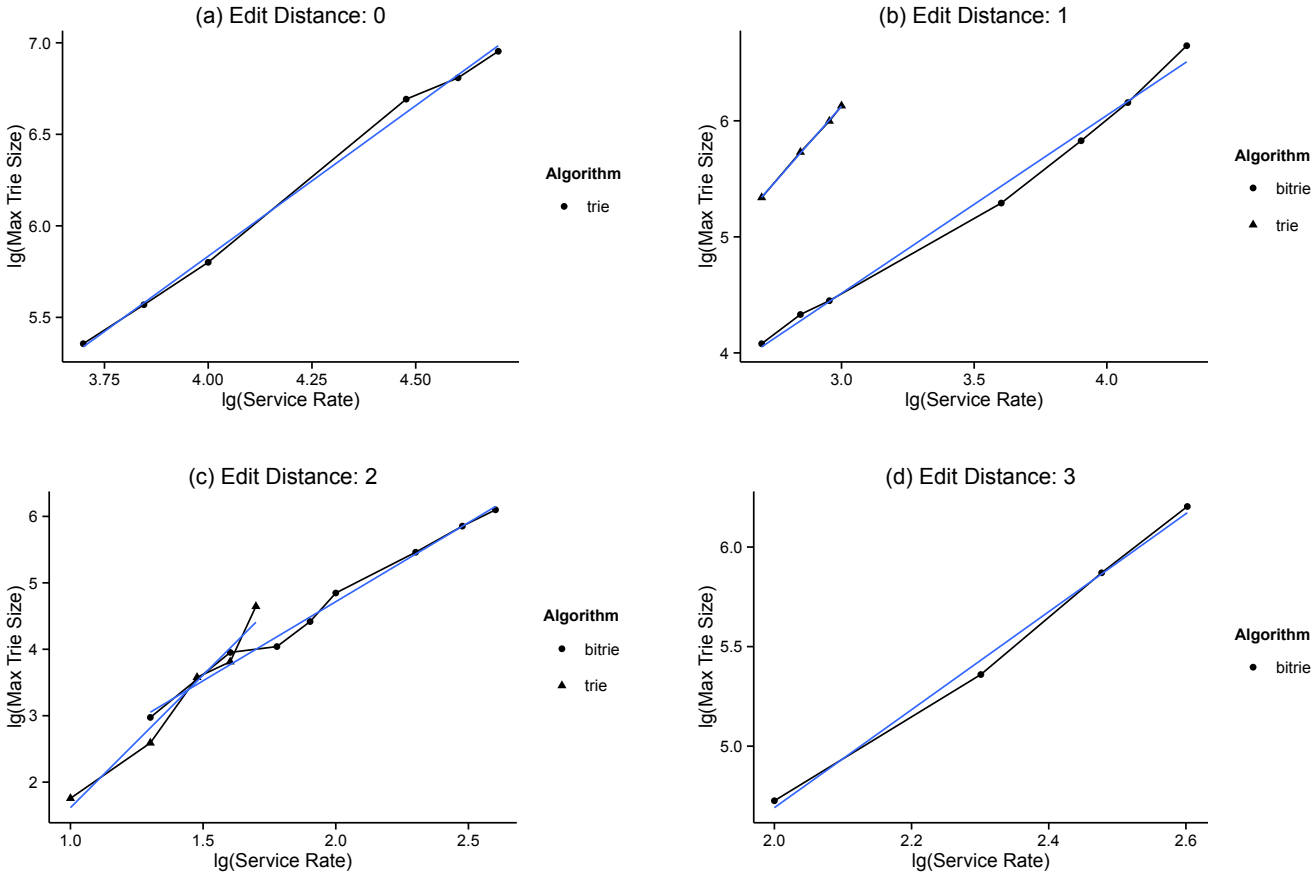

Figure 5: Comparison of trie and bi-trie with DBLP Authors

[13] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):976–991, 2008.

[14] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of SIGMOD*, pages 1033–1044. ACM, 2011.

[15] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proceedings of the Annual Symposium on the Theory of Computing*, pages 218–223. ACM, 1975.

[16] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. In *Proceedings of VLDB*, pages 1219–1230, 2010.

[17] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of SIGMOD*, pages 85–96. ACM, 2012.

[18] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. VChunkJoin: An efficient algorithm for edit similarity joins. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1916–1929, 2013.

[19] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In *Proceedings of VLDB*, pages 933–944, 2008.

[20] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.