Near Neighbor Join

Herald Kllapi [#+1], Boulos Harb [*2], Cong Yu [*3]

[#] *Dept. of Informatics and Telecommunications, University of Athens*
Panepistimiopolis, Ilissia Athens 15784, Greece
[1] herald@di.uoa.gr
[+] Work done while at Google Research.

[*] *Google Research,*
75 Ninth Avenue, New York, NY 10011, USA
[2] harb@google.com
[3] congyu@google.com

***Abstract*—An increasing number of Web applications such as friends recommendation depend on the ability to join objects at scale. The traditional approach taken is *nearest neighbor join* (also called *similarity join*), whose goal is to find, based on a given join function, the *closest* set of objects or *all* the objects within a distance threshold to each object in the input. The scalability of techniques utilizing this approach often depends on the characteristics of the objects and the join function. However, many real-world join functions are intricately engineered and constantly evolving, which makes the design of white-box methods that rely on understanding the join function impractical. Finding a technique that can join extremely large number of objects with complex join functions has always been a tough challenge.**

In this paper, we propose a practical alternative approach called *near neighbor join* that, although does not find the closest neighbors, finds *close* neighbors, and can do so at extremely large scale when the join functions are complex. In particular, we design and implement a super-scalable system we name SAJ that is capable of *best-effort* joining of billions of objects for complex functions. Extensive experimental analysis over real-world large datasets shows that SAJ is scalable and generates good results.

I. Introduction

Join has become one of the most important operations for Web applications. For example, to provide users with recommendations, social networking sites routinely compare the behaviors of hundreds of millions of users [1] to identify, for each user, a set of similar users. Further, in many search applications (e.g., [2]), it is often desirable to showcase additional results among billions of candidates that are related to those already returned.

The join functions at the core of these applications are often very complex: they go beyond the database style θ-joins or the set-similarity style joins. For example, in Google Places, the similarity of two places are computed based on combinations of spatial features and content features. For WebTables used in Table Search [3], the similarity function employs a multi-kernel SVM machine learning algorithm. Neither function is easy to analyze.

Furthermore, the needs of such applications change and the join functions are constantly evolving, which makes it impractical to use a system that heavily depends on function-specific optimizations. This complex nature of real-world join functions makes the join operation especially challenging to perform at large scale due to its inherently quadratic nature, i.e., there is no easy way to partition the objects such that only objects within the same partition need to be compared.

Fortunately, unlike database joins where exact answers are required, many Web applications accept join results as long as they are near. *For a given object, missing some or even all of the objects that are nearest to it, is often tolerable if the objects being returned are almost as near to it as those that are nearest.* Inability to scale to the amount of data those applications must process, however, is not an option. In fact, a key objective for all such applications is to balance the result accuracy and the available machine resources while processing the data in its entirety.

In this paper, we introduce SAJ[1], a Scalable Approximate Join system that performs *near neighbor join* of billions of objects of *any type* with a broader set of complex join functions, where the only expectation on the join function is that it satisfies the triangle inequality[2]. More specifically, SAJ aims to solve the following problem: Given (1) a set I of N objects of type T, where N can be billions; (2) a complex join function $F_J : T \times T \to \mathbb{R}$ that takes two objects in I and returns their similarity; and (3) resource constraints (specified as machine per-task computation capacity and number of machines). For all $o \in I$, find k objects in I that are similar to o according to F_J without violating the machine constraints.

As with many other recent parallel computation systems, SAJ adopts the MapReduce programming model [4]. At a high level, SAJ operates in two distinct multi-iteration phases. In the initial *Bottom-Up* (BU) phase, the set of input objects are iteratively partitioned and clustered within each partition to produce a successively smaller set of representatives. Each representative is associated with a set of objects that are similar to it within the partitions in the previous iteration. In the following *Top-Down* (TD) phase, at each iteration the most similar pairs of representatives are selected to *guide* the comparison of objects they represent in the upcoming iteration.

[1] In Arabic, *Saj* is a form of rhymed prose known for its evenness, a characteristic that our system strives for.

[2] In fact, even the triangle inequality of the join function is not a strict requirement within SAJ, it is only needed if a certain level of quality is to be expected, see Section V.

To achieve true scalability, SAJ respects the resource constraints in two critical aspects. First, SAJ strictly adheres to *machine per-task capacity* by controlling the partition size so that it is never larger than the number of objects each machine can handle. Second, SAJ allows developers during the TD phase to easily adjust the accuracy requirements in order to satisfy the resource constraint dictated by the number of machines. Because of these design principles, in one of our scalability experiments, SAJ completed a near-k (where $k = 20$) join for 1 *billion* objects within 20 hours.

To the best of our knowledge, our system is the first attempt at super large scale join without detailed knowledge of the join function. Our specific contributions are:

- We propose a novel top-down scalable algorithm for selectively comparing promising object pairs starting with a small set of representative objects that are chosen based on well-known theoretical work.
- Based on the top-down approach, we build an end-to-end join system capable of processing near neighbor joins over billions of objects without requiring detailed knowledge about the join function.
- We provide algorithmic analysis to illustrate how our system scales while conforming to the resource constraints, and theoretical analysis of the quality expectation.
- We conducted extensive experimental analysis over large scale datasets to demonstrate the system's scalability.

The rest of the paper is organized as follows. Section II presents the related work. Section III introduces the basic terminologies we use, the formal problem definition, and an overview of the SAJ system. Section IV describes the technical components of SAJ. Analysis of the algorithms and the experiments are described in Sections V and VI, respectively. Finally, we conclude in Section VII.

II. Related Work

Similarity join has been studied extensively in recent literature. The work by Okcan and Riedewald [5] describes a cost model for analyzing database-style θ-joins, based on which an optimal join plan can be selected. The work by Vernica et. al. [6] is one of the first to propose a MapReduce-based framework for joining large data sets using set similarity functions. Their approach is to leverage the nature of set similarity functions to prune away a large number of pairs before the remaining pairs are compared in the final reduce phase. More recently, Lu et. al. [7] apply similar pruning techniques to joining objects in n-dimensional spaces using MapReduce. Similar to [6], [7], there are a number of studies on scalable similarity join using parallel and/or p2p techniques [8], [9], [10], [7], [11], [12], [13], [14]. Those proposed techniques deal with join functions in two main categories: (i) set similarity style joins where the objects are represented as sets (e.g., Jaccard), or (ii) join functions that are designed for spatial objects in n-dimensional vector space, such as L_p distance.

Our work distinguishes itself in three main aspects. First, all prior works use knowledge about the join function to perform pruning and provide exact results. SAJ, while assuming triangle inequality for improved result quality, assumes little about the join function and produces best-effort results. Second, objects in those prior works must be represented either as a set or a multi-dimensional point, while SAJ makes no assumption about how object can be represented. Third, the scalability of some of these studies follows from adopting a high similarity threshold, hence they cannot guarantee that k neighbors are found for every object. The tradeoff is that SAJ only provides best-effort results, which are reasonable for real world applications that require true scalability and can tolerate non-optimality.

Scalable (exact and approximate) similarity joins for known join functions have been studied extensively in non-parallel contexts [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. The join functions being considered include edit distance, set similarity, cosine similarity, and L_p distance. Again, they all use knowledge about the join functions to prune candidate pairs and avoid unnecessary comparisons. For general similarity join, [26] introduces a partition based join algorithm that only requires the join function to have the metric property. The algorithm, however, is designed to discover object pairs within ϵ distance of each other instead of finding k neighbors of each object. LSH has been effective for both similarity search [27] and similarity join [28]; however, it is often difficult to find the right hash functions.

There are some works that consider the problem of incremental kNN join [29]. Since our framework is implemented on top of MapReduce, we focus on designing a system that scales to billions of objects using batch processing of read-only datasets and do not consider this case.

Our work is related to techniques developed for kNN search [30], whose hashing techniques can potentially be leveraged in SAJ. Also focusing on search, there are several works that use a similar notion of representative objects to build a tree and prune the search space based on the triangle innequality property [31], [32], [33]. Finally, our Bottom-Up algorithm adopts the streaming clustering work in [34].

III. Preliminaries & SAJ Overview

Notation	Semantics
$I, N = \|I\|$	The set of input objects and its size
k	Desired number of near neighbors per object
n	Maximum number of objects a machine can *manipulate* in a single task, $n \ll N$
m	Number of clusters per local partition, $m < n$
P	Number of top object pairs maintained for each TD iteration, $P \ll N^2$
p	Maximum number of object pairs sent to a machine in each TopP iteration, $p > P$
F_J	Required user-provided join function: $F_J : I \times I \rightarrow \mathbb{R}$
F_P	Optional partition function: $F_P : I \rightarrow$ `string`
F_R	Optional pair ranking function: $F_R : (I, I) \times (I, I) \rightarrow \mathbb{R}$

TABLE I
ADOPTED NOTATION.

Table I lists the notation we use. In particular, n is a key system parameter determined by the number of objects a single machine task can reasonably *manipulate*. That is: (i) the available memory on a single machine should be able to simultaneously hold n objects; and, (ii) the task of all-pairs comparison of n objects, which is an $O(n^2)$ operation, should complete in reasonable amount of time. For objects that are large (i.e., tens of megabytes), n is restricted by (i), and for objects that are small, n is restricted by (ii). In practice, n is either provided by the user directly or estimated through some simple sampling process. Another key system parameter is P, which controls the number of top pairs processed in each iteration by the TD phase (cf. Section IV-B). It is determined by the number of available machines and the user's quality requirements: increasing P leads to better near neighbors for each object, but demands more machines and longer execution time. It is therefore a critical knob that balances result quality and resource consumption. The remaining notation is self-explanatory, and readers are encouraged to refer to Table I throughout the paper.

We build SAJ on top of MapReduce [4] which is a widely-adopted, shared-nothing parallel programming paradigm with both proprietary [4] and open source [35] implementations. The MapReduce paradigm is well suited for large scale offline analysis tasks, but writing native programs can be cumbersome when multiple Map and Reduce operations need to be chained together. To ease the developer's burden, high level languages are often used such as Sawzall [36], Pig [37], and Flume [38]. SAJ adopts the Flume language.

Our goal for SAJ is a super-scalable system capable of performing joins for billions of objects with a user provided complex join function, and running on commodity machines that are common to MapReduce clusters. Allowing user provided complex join functions introduces a significant new challenge: *we can no longer rely on specific pruning techniques* that are key to previous solutions such as prefix filtering for set similarity joins [6]. Using only commodity machines means we are required to control the amount of computation each machine performs regardless of the distribution of the input data. Performing *nearest* neighbor join under the above two constraints is infeasible because of the quadratic nature of the join operation. Propitiously, real-world applications do not require optimal solutions and prefer best-effort solutions that can scale, which leads us to the *near* neighbor join approach.

A. Problem Definition

We are given input $I = \{o_i\}_{i=1}^{N}$ where N is very large (e.g. billions of objects); a user-constructed join function $F_J : I \times I \rightarrow \mathbb{R}$ that computes a distance between any two objects in I, and that we assume no knowledge except that F_J satisfies the triangle inequality property (for SAJ to provide quality expectation); a parameter k indicating the desired number of neighbors per object; and machine resource constraints. The two key resource constraints are: 1) the number of objects each machine can be expected to perform an all-pairs comparison on; and 2) the maximum number of records each Shuffle phase in the program is able to handle, which can be derived from the number of machines available in the cluster.

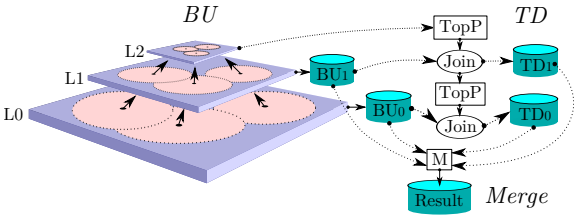

Fig. 1. Overview of SAJ with Three Levels.

We produce output $O = \{(o_i \rightarrow R_i)\}_{i=1}^{N}$ where R_i is the set of k objects in I we discover to be near o_i according to F_J. For each o_i, let R_i^{nearest} be the top-k nearest neighbors of o_i; i.e., $\forall (j,k), o_j \notin R_i^{\text{nearest}}, o_k \in R_i^{\text{nearest}}, F_J(o_i, o_j) \geq F_J(o_i, o_k)$. Let $E_i = \text{AVG}_{o \in R_i}(F_J(o_i, o)) - \text{AVG}_{o \in R_i^{\text{nearest}}}(F_J(o_i, o))$ be the average distance error of R_i. Our system attempts to reduce $\text{AVG}_i(E_i)$ in a *best-effort* fashion while respecting the resource constraints.

B. Intuition and System Overview

The main goal of SAJ is to provide a *super scalable* join solution that can be applied to user provided *complex* join functions, and obtain results that, despite being *best-effort*, are significantly better than a random partition approach would obtain and close to the optimal results.

To achieve higher quality expectation than a random approach, we assume the join functions to satisfy the triangle inequality property (similar to previous approaches [7]). Given that, we make the following observation. Given five objects $o_{x_1}, o_{x_2}, o_m, o_{y_1}, o_{y_2}$, if $F_J(o_{x_1}, o_m) < F_J(o_{x_2}, o_m)$ and $F_J(o_{y_1}, o_m) < F_J(o_{y_2}, o_m)$, then $\text{Prob}[F_J(o_{x_1}, o_{y_1}) < F_J(o_{x_2}, o_{y_2})]$ is greater than $\text{Prob}[F_J(o_{x_2}, o_{y_2}) < F_J(o_{x_1}, o_{y_1})]$. In other words, two objects are more likely to be similar if they are both similar to some common object. We found this to hold for many real-world similarities even those complex ones provided by the users. This observation leads to our overall strategy: *increase the probability of comparing object pairs that are more likely to yield smaller distances by putting objects that are similar to common objects on the same machine*.

As illustrated in Figure 1, SAJ implements the above strategy using three distinct phases: BU, TD, and Merge. The BU phase (cf. Section IV-A) adopts the Divide-and-Conquer strategy of [34]. It starts by partitioning objects in I, either randomly or based on a custom partition function, into sets small enough to fit on a single machine. Objects in each partition are clustered into a small set of representatives that are sent to the next iteration level as input. The process repeats until the set of representatives fit on one machine, at which point an all-pairs comparison of the final set of representatives is performed to conclude the BU phase. At the end of this phase, in addition to the cluster representatives, a preliminary list of near-k neighbors ($k \leq n$) is also computed for each object in I as a result of the clustering.

Types	Definitions
`Object`	user provided
`ID`	user provided
`Obj`	<ID id, Object raw_obj, Int level, Int weight>
`Pair`	<ID from_id, ID to_id, Double distance>
`SajObj`	<Obj obj, ID id_rep, List<Pair> pairs>

TABLE II
DEFINITIONS OF DATA TYPES USED BY SAJ.

The core of SAJ is the multi-iteration TD phase (cf. Section IV-B). At each iteration, top representative pairs are chosen based on their similarities through a distributed TopP computation. The chosen pairs are then used to fetch candidate representative pairs from the corresponding BU results in parallel to update each other's nearest neighbor lists and to generate the top pairs for the next iteration. The process continues until all levels of representatives are consumed and the final comparisons are performed on the non-representative objects. The key idea of SAJ is to strictly control the number of comparisons regardless of the overall input size, thus making the system super scalable and immune to likely input data skew. At the end of this phase, a subset of the input objects have their near-k neighbors from the BU phase updated to a better list.

Finally, the Merge phase (cf. Section IV-C) removes obsolete neighbors for objects whose neighbors have been updated in the TD phase. While technically simple, this phase is important to the end-to-end pipeline, which is only valuable if it returns a single list for each object.

IV. THE SAJ SYSTEM

For ease of explanation, we first define the main data types adopted by SAJ in Table II. Among the types, `Object` and `ID` are types for the raw object and its identifier, respectively, as provided by the user. `Obj` is a wrapper that defines the basic object representation in SAJ: it contains the raw object (`raw_obj`), a globally unique ID (`id`), a `level` (for internal bookkeeping), and the number of objects assigned to it (`weight`). `Pair` represents a pair of objects, which consists of identifiers of the two objects and a `distance` between them[3], which is computed from the user provided F_J. Finally, `SajObj` represents the rich information associated with each object as it flows through the pipeline, including its representative (i.e., `id_rep`) at a higher level and the current list of near neighbors (`pairs`).

In the rest of this section, we describe the three phases of SAJ in details along with the example shown in Figure 2(A). In our example, the input dataset has $N = 16$ objects that are divided into 4 clusters. We draw objects in different clusters with a different shape, e.g., the bottom left cluster is the *square* cluster and the objects within it are named s_1 through s_4. The other three clusters are named *dot*, *cross*, *triangle*, respectively.

[3]For simplicity, we assume F_J is directional, but SAJ works equally on non-directional join functions.

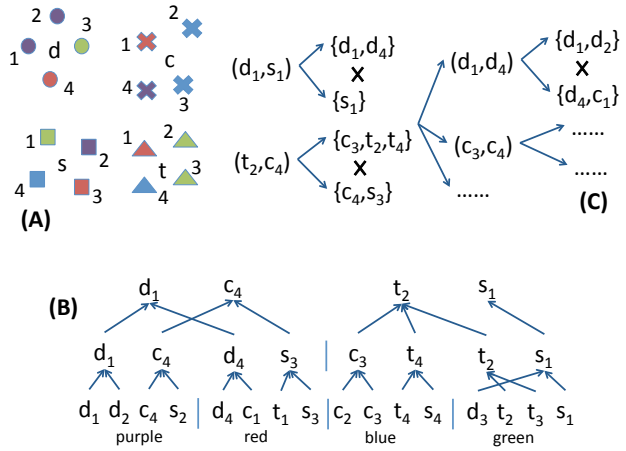

Fig. 2. An End-to-End Example: (A) The input dataset with 16 objects, roughly divided into 4 clusters (as marked by shape); (B) BU phase; (C) TD phase.

We use $n = 4, m = 2$, i.e., each partition can handle 4 objects and produce 2 representatives, and set $k = 2, P = 2$.

A. The Bottom-Up Phase

Algorithm 1 The BU Pipeline.

Require: I, the input dataset; N, the estimated size of I; k, n, m, F_P, F_J as described in Table I.

1: Set<`Object`> input = Open(I);
2: Set<`SajObj`> current = **Convert**(input);
3: $level$ = 0, N' = N;
4: **while** $N' > n$ **do**
5: MapStream<`String`,`SajObj`> shuffled = **Shuffle**(current, N', n, F_P));
6: Map<`String`,`SajObj`> clustered = **PartitionCluster**(shuffled, m, k, F_J);
7: current = **Materialize**(clustered, $level$);
8: $level$++, N' = N' / (n/m);
9: **end while**

Return: $current$, the final set of representative objects with size $\leq n$; $level$, the final level at which BU phase stops, $BU_0, BU_1, \ldots, BU_{level}$, sets of `SajObj` objects that are materialized at each BU level.

Algorithm 1 illustrates the overall pipeline of the multi-iteration BU phase, which is based on [34]. The algorithm starts with converting the input `Objects` into `SajObjs` in a parallel fashion (Line 2), setting the default values and extracting a globally unique identifier for each object. After that, the algorithm proceeds iteratively. Each iteration (Lines 5-9) splits the objects into partitions, either randomly or using F_P, and from each partition extracts a set of representative objects (which are sent as input to the next iteration) and writes them to the underlying distributed file system along with the preliminary set of near-k neighbors. The BU phase concludes when the number of representative objects falls below n.

The example of Figure 2 will help illustrate. The objects are first grouped into partitions (assume F_J is random). The partitions are indicated by the color (*purple*, *red*, *green*, or *blue*) in Figure 2(A), as well as the groups in the bottom level in Figure 2(B). Given the input partitions, the first iteration of the BU phase performs clustering on each partition in parallel and produces 2 representatives, as indicated by the middle level in Figure 2(B). As to be described in Section IV-A, two pieces of data are produced for each object. First, a preliminary list of near neighbors is computed for each object regardless whether it is selected as a representative. Second, one of the representatives, which can be the object itself, is chosen to represent the object. For example, object d_2, after the local clustering, will have its near-k list populated with $\{d_1, c_4\}$ and be associated with representative d_1. Both are stored into the result for this BU iteration level (BU_0). The first iteration produces 8 representative objects, which are further partitioned and clustered by the second iteration to produce the final set of representatives, $\{d_1, c_4, t_2, s_1\}$, as shown in the top level of Figure 2(B).

At the core of the BU phase is *PartitionCluster* shown in Algorithm 2. It is executed during the reduce phase of the MapReduce framework and operates on groups (i.e., streams) of `SajObj`s grouped by the *Shuffle* function. Each stream is sequentially divided into partitions of n (or less) objects each. This further partitioning is necessary: while F_P aims to generate groups of size n, larger groups are common due to skew in the data or improperly design F_P. Without further partitioning, a machine can easily be overwhelmed by a large group. For each partition, *PartitionCluster* performs two tasks. First, it performs an all pairs comparison among the objects using F_J. Since each object is compared to $(n-1)$ other objects, a preliminary near-k neighbors are stored as a side effect (we assume $k \ll n$). Second, it applies a state-of-art clustering algorithm (e.g., Hierarchical Agglomerative Clustering [39]) or one provided by the user, and identifies a set of m cluster centers. The centers are chosen from the input objects as being closest to all other objects on average (i.e., medoids). Each non-center object is assigned to a representative, which is the cluster center closest to it.

Finally, the *Materialize* function writes the `SajObj` objects, along with their representatives and preliminary list of near neighbors, to the level-specific output file (e.g., BU_0) to be used in the TD phase later. It also emits the set of representative `SajObj` objects to serve as the input to the next iteration.

The final result of the BU phase is conceptually a tree with $\log_{n/m}(N)$ levels and each object assigned to a representative in the next level up. We note that each iteration produces an output of size (n/m) times smaller than its input and most datasets can be processed in a small number of iterations with reasonably settings of n and m.

a) Further Discussion: When a user provided F_P is adopted for partition, it is possible extreme data skew can occur where significantly more than n objects are grouped together and sent to the same reducer. Although PartitionCluster ensures that the all pairs comparison is always performed on no more than n objects at one time, the reducer can still become overwhelmed having to handle so many objects. When SAJ detects this, it uses an extra MapReduce round in Shuffle to split large partitions into random smaller partitions before distributing them. We also use a similar technique to combine small partitions to ensure that each partition has more than $(n/2)$ objects. The details are omitted due to space.

The partition function creates groups of objects, possibly greater than n in size due to data skew. We relax the constraint

Algorithm 2 Bottom-Up Functions.

Shuffle($objects$, N, n, F_P)
Require: $objects$, a set with objects of type `SajObj`;
1: Map<`String`,`SajObj`> shuffled;
2: **for each** $o \in objects$ **do**
3: **if** $F_P \neq$ empty **then**
4: shuffled.put(F_P(o.`obj.raw_obj`), o);
5: **else**
6: shuffled.put(Random[0, $\lceil N/n \rceil$), o);
7: **end if**
8: **end for**
9: return shuffled;

PartitionCluster($shuffled$, m, k, F_J)
Require: $shuffled$, a map with stream of objects of type `SajObj`;
1: **for each** $stream \in shuffled$ **do**
2: $count$ = 0;
3: Set<`SajObj`> partition = $\emptyset$
4: **while** $stream$.has_next() **do**
5: **while** $count < n$ **do**
6: partition.add($stream$.next());
7: **end while**
8: Matrix<`ID`, `ID`> matrix = $\emptyset$; // distance matrix
9: **for** $0 \leq i, j \leq |partition|, i \neq j$ **do**
10: from = partition[i], to = partition[j];
11: d = F_J(from.`obj.raw_obj`, to.`obj.raw_obj`);
12: matrix[from.`obj.id`, to.`obj.id`] = d;
13: Insert(from.`pairs`, `Pair`(from.`obj.id`, to.`obj.id`, d));
14: **if** |from.`pairs`| $> k$ **then**
15: RemoveFurthest(from.`pairs`);
16: **end if**
17: **end for**
18: C = InMemoryCluster(matrix, partition, m);
19: **for each** o $\in$ partition **do**
20: AssignClosestCenterOrSelf(o.`id_rep`, C);
21: **EMIT**(o);
22: **end for**
23: $count = 0$, partition = $\emptyset$; // reset for the next partition
24: **end while**
25: **end for**

Materialize($clustered$, $level$): `MapFunc` applied to one record at a time.
Require: $clustered$, a set with objects of type `SajObj`;
1: **for each** $o \in clustered$ **do**
2: **if** o.`obj.id` == o.`id_rep` **then**
3: **EMIT**(o); // this object is a representative
4: **end if**
5: **WriteToDistributedFileSystem**(BU_{level}, o);
6: **end for**

of each partition being exactly n objects and guarantee that each partition is no less than $\frac{n}{2}$ objects and no more than n objects. To achieve this, we look ahead the stream to and if there are more than $(1+\frac{1}{2})n$ objects, we group the first n objects. If there are less than $(1+\frac{1}{2})n$, the objects are divided into two equal groups. This way, the last two partitions will have $\frac{(1+b)\cdot n}{2}$ objects, with $0 < b < \frac{1}{2}$.

B. The Top-Down Phase

During the BU phase, two objects are compared if and only if they fall into the same partition during one of the iterations. As a result, two highly similar objects may not be matched. Comparing all objects across different partitions, however, is infeasible since the number of partitions can be huge. The goal of SAJ is therefore refining the near neighbors computed from the BU phase by selectively comparing objects that are likely to be similar. To achieve that goal, the key technical contribution of SAJ is to *dynamically compute top-P pairs and guide the comparisons of objects*. A set of closest pairs of representatives can therefore be used as a guide to compare the objects they represent. The challenge is how to compute and apply this guide efficiently and in a scalable way: the TD phase is designed to address this.

Algorithm 3 The Top-Down Pipeline

Require: $BU_0, BU_1, \ldots, BU_f$, the level-by-level objects of type `SajObj` produced by the BU phase; P, p, F_R, F_J as described in Table I.

// we first declare some variables used in each iteration

1: Map<`String`,`SajObj`> from_cands, to_cands; // candidate objects from different partitions to be compared.

2: Set<`Pair`> bu_pairs, td_pairs; // pairs already computed by BU or being computed by TD .

// get the top-p pairs from the final BU level

3: Set<`Pair`> top = **InMemoryTopP**(BU_f, P, F_J, F_R);

4: **for** $level = (f\text{-}1) \rightarrow 0$ **do**

5: (from_cands, to_cands, bu_pairs) = **GenerateCandidates**(BU_{level}, top);

// define type Set<`SajObj`> as `SSO` for convenience

6: Map<`String`,(`SSO`, `SSO`)> grouped = Join(from_cands, to_cands);

// CompareAndWrite writes to TD_{level}

7: td_pairs = **CompareAndWrite**(grouped, $F_J, level$);

8: Set<`Pair`> all_pairs = bu_pairs ∪ td_pairs;

9: **if** $level > 0$ **then**

10: top = **DistributedTopP**(all_pairs, P, p, F_R);

11: **end if**

12: **end for**

Return: $TD_0, TD_1, \ldots, TD_f$, sets of `SajObj` objects that are materialized at each TD level.

Algorithm 3 illustrates the overall flow of the TD phase. It starts by computing the initial set of top-P pairs from the top-level BU results (Line 3). The rest of the TD pipeline is executed in multiple iterations from top to bottom, i.e., in reverse order of the BU pipeline. At each iteration, objects belonging to different representatives and different partitions are compared under the guidance of the top-P pairs computed in the previous iteration. Those comparisons lead to refinements of the near neighbors of the objects that are compared, which were written out for the Merge phase. To guide the next iteration, each iteration produces a new set of top pairs in a distributed fashion using DistributedTopP shown in Algorithm 4.

The example of Figure 2 will help illustrate. The phase starts by performing a pair-wise comparison on the final set of representatives and produces the top pairs based on their similarities (as computed by F_J). Then, as shown in Figure 2(C), for both objects in a top pair, we gather the objects they represent from the BU results in a distributed fashion as described in Section IV-B. For example, the pair (t_2, c_4) guides us to gather $\{c_3, t_2, t_4\}$ and $\{c_4, s_3\}$. Objects gathered based on the same top pair are compared against each other to produce a new set of candidate pairs. From the candidate pairs generated across all current top pairs, the DistributedTopP selects the next top pairs to guide the next iteration. The process continues until all the BU results are processed. The effect of the TD phase can be shown through the trace of object d_2 in our example. In the first BU iteration, d_2 is compared with d_1, c_4, s_2 and populates its near-k list with $\{d_1, c_4\}$. Since d_2 is not selected as a representative, it is never seen again, and without the TD phase, this is the best result we can get. During the TD phase, since (d_1, d_4) is the top pair in the second iteration, under its guidance, d_2 is further compared with d_4, c_1, leading to d_4 being added to the near-k list for d_2. The effects of P can also be illustrated in this example. As P increases, the pair (d_1, s_1) will eventually be selected as a top pair, and with that, d_2 will be compared with d_3, s_1, leading to d_3 being added to its final near-k list.

Each iteration of the TD phase uses two operations, *GenerateCandidates* and *CompareAndWrite* shown in Algorithm 5. *GenerateCandidates* is applied to the objects computed by the BU phase at the corresponding level. The goal is to identify objects that are worthy of comparison. For each input object, it performs the following: if the representative of the object matches at least one top pair, the object itself is emitted as the `from` candidate, the `to` candidate, or both, depending how the representative matches the pair. The key is chosen such that, for each pair in *top*, we can group together all and only those objects that are represented by objects in the pair. We also emit the near neighbor pairs computed by the BU phase. It is necessary to retain those pairs because some of the neighbors may be very close to the object, and because those neighbors and the input object itself are themselves representatives of objects in the level down.

The choice of the emit key, however, is not simple. Our goal is that, for each pair in *top*, group together all and only those objects that are represented by objects in the pair. This means the key needs to be unique to each pair even though we don't have control over the vocabulary of the object ID[4]. A simple concatenation of the two object IDs do not work. For example, if the separator is "`|`", then consider two objects involved in the pair with IDs "`abc|`" for `from` and "`|xyz`" for

[4]Enforcing a vocabulary can be impractical in real applications.

Algorithm 4 InMemory & Distributed Top-P Pipelines.

InMemoryTopP(O, P, F_J, F_R): Computing the top-P pairs.

Require: O, the in-memory set with objects of type `SajObj`; P, F_J, F_R as described in Table I.

```
Set<Pair> pairs = ∅
for i, j = 0 → (|O| − 1), i ≠ j do
    Pair pair = (O[i].obj.id, O[j].obj.id,
                 F_J(O[i].obj.raw_obj, O[j].obj.raw_obj));
    pairs.Add(pair);
end for
top_pairs = TopP(pairs, P, F_R)
```

Return: top_pairs

DistributedTopP($pairs, p, P, F_R$): Computing the top-P pairs using multiple parallel jobs.

Require: $pairs$, the potentially large set of object pairs of type Set<`Pair`>; p, P, F_R as described in Table I.

```
num_grps = estimated_size(pairs)/p ;
Set<Pair> top; // the intermediate groups of top-P pairs, each of which contains ≤ P pairs.
top = pairs;
while num_grps > 1 do
    Set<Pair> new_top;
    for 0 ≤ g < num_grps do
        new_top.add(TopP(top_g, P, F_R)); // Run TopP on g part of top
    end for
    num_grps = Max(1, num_grps/p);
    top = new_top;
end while
```

Return: top

TopP($pairs, P, F_R$): Basic routine for Top-P computation.

Require: $pairs$, Set or Stream of object pairs of type `Pair`; P, F_R as described in Table I.

```
static Heap<Pair> top_pairs = Heap(∅, F_R); // A heap for keeping P closest pairs, where the pairs are scored by F_R.
for pair ∈ pairs do
    top_pairs.Add(pair);
    if top_pairs.size() > P then
        top_pairs.RemoveFurthest();
    end if
end for
```

Return: Set(top_pairs)

Algorithm 5 Top-Down Functions.

GenerateCandidates(BU_{level}, top)

Require: BU_{level}, a set of Bottom-Up result of type `SajObj`; top, the top-P closest pairs (of type `Pair`) computed from the previous iteration.

```
static initialized = false; // per machine static variable.
static Map<String, List<Pair>> from, to; // per machine indices for object IDs and their top pairs.
if not initialized then
    for each p ∈ top do
        from.push(p.id_from, p);
        to.push(p.id_to, p);
    end for
    initialized = true;
end if
Map<String,SajObj> from_cands, to_cands;
Set<Pair> bu_pairs;
for each o ∈ BU_level do
    if from.contains(o.id_rep) or to.contains(o.id_rep) then
        for each p_from ∈ from.get(o.id_rep) do
            from_cands.put(Key(p_from), o);
        end for
        for each p_to ∈ to.get(o.id_rep) do
            to_cands.put(Key(p_to), o);
        end for
    else
        for each p_bu ∈ o.pairs do
            bu_pairs.add(p_bu);
        end for
    end if
end for
```

Return: from_cands, to_cands, bu_pairs;

CompareAndWrite($grouped$, $F_J, level$)

Require: $grouped$, a set of pairs $(from\_cands, to\_cands)$, two streams of type Stream <`SajObj`> for objects represented by objects in one of the top-P pairs; $F_J, level$, as described in Algorithm 3.

```
Set<SajObj> updated; // tracking the objects whose top-P pairs have been updated.
Set<Pair> td_pairs;
for each (from_cands, to_cands) ∈ grouped do
    for each o_f ∈ from_cands, o_t ∈ to_cands do
        Pair p_td = (o_f.obj.id, o_t.obj.id,
                     F_J(o_f.obj.raw_obj, o_t.obj.raw_obj));
        if UpdateRelatedPairs(o_f.pairs, p_td) then
            updated.add(o_f); // updated o_f's near-k related pairs
        end if
        if UpdateRelatedPairs(o_t.pairs, p_td) then
            updated.add(o_t); // updated o_t's near-k related pairs
        end if
        td_pairs.add(p_td);
    end for
end for
WriteToDistributedFileSystem(TD_level, updated);
```

Return: td_pairs;

`to`. Simple concatenation gives us a key of "`abc|||xyz`", which is indistinguishable from the key for two objects with IDs "`abc||`" and "`xyz`". The solution we came up with is to assign the key as "`len:id_from|id_to`". With this scheme, we will generate two distinct keys for the above example, "`4:abc|||xyz`" and "`5:abc|||xyz`". We can show that this scheme generates keys unique to each pair regardless of the vocabulary of the object IDs.

CompareAndWrite performs the actual comparisons for each top pair and *this is where the quality improvements over the BU results happen*. The inputs are two streams of objects, containing objects represented by `from` and `to` of the top pair being considered, respectively. An all-pair comparison is performed on those two streams to produce new pairs, which are subsequently emitted to be considered in the top-P. During the comparison, we update the near neighbors of the two objects being compared if they are closer to each other than the existing neighbors they have. The updated objects are written out as the TD results.

All the produced pairs are union'ed to produce the full set of pairs for the level. This set contains a much larger number of pairs than any single machine can handle. We need to compute the top-P closest pairs from this set efficiently and in a scalable way, which is accomplished by the *DistributedTopP*

function.

C. The Merge Phase

The Merge phase produces the final near-$k$neighbors for each object by merging the results from the BU and TD phases. This step is necessary because: 1) some objects may not be updated during the TD phase (i.e., the TD results may not be complete), and 2) a single object may be updated multiple times in TD with different neighbors. At the core of Merge is a function, which consolidates the multiple near neighbor lists of the same object in a streaming fashion.

V. Theoretical Analysis

A. Complexity Analysis

In this section we show the number of MapReduce iterations is $O(\log(N/n)\log((kN + n^2P)/p)/\log(n/m)\log(p/P))$, and we bound the work done by the tasks in each of the BU, TD and Merge phases.

MR Iterations. Let L be the number of MapReduce iterations in the BU phase. Suppose at each iteration i we partition the input from the previous iteration into L_i partitions. Each partition R has size $\lfloor mL_{i-1}/L_i \rfloor$ since m representatives from each of the L_{i-1} partitions are sent to the next level. The number L_i is chosen such that $n/2 \leq |R| \leq n$ ensuring that the partitions in that level are large but fit in one task (i.e. do not exceed n). This implies that at each iteration, we reduce the size of the input data by a factor $c = L_{i-1}/L_i \geq n/(2m)$. The BU phase stops at the first iteration whose input size is $\leq n$ (and *w.l.o.g.* assume it is at least $n/2$). Hence, the number of iterations is at most:

$$L = \log_c(N) - \log_c(n/2) = \log_c(2N/n) \leq \frac{\log(2N/n)}{\log(n/(2m))} \ .$$

For example, if $N = 10^9$, $n = 10^3$, and $m = 10$, then $L = 4$.

Now, each iteration i in the TD phase consumes the output of exactly one iteration in the BU phase, and produces $O(E)$ object pairs, where $E = (nL_{i-1} - P)k + n^2P)$. These account for both the pairs produced by the comparisons and the near neighbors of the objects. The top P of these pairs are computed via a MapReduce tree similar to that of the BU phase. Using analysis similar to the above and recalling the definition of p, the number of iterations in this MR tree is $O(\log(E/p)/\log(p/P))$. Hence, using $L_i \leq N/n$, the total number of iterations in the TD phase is,

$$O\left(\frac{\log\prod_{i=1}^{L} E/p}{\log(p/P)}\right) = O\left(\frac{L\log((kN + n^2P)/p)}{\log(p/P)}\right),$$

For illustration, going back to our example, if $P = 10^7$, $p = 10^9$, and $k = 20$, then the number of iterations in the TD phase (including DistributedTopP) is 16, among which only the first few iterations are expensive.

Time Complexity. In the BU phase, each Reducer task reads n objects, clusters them and discovers related pairs in $O(n^2)$ time. It then writes the m representatives and the n clustered objects (including the k near neighbors embedded in each) in $O(n + m)$ time. Thus, the running time of each task in this phase is bounded by $O(n^2)$. Let M be the number of machines. The total running time of the BU Reduce Phase is then $O(\sum_{i=0}^{L}\lceil L_i/M\rceil n^2)$ which is $O(n^2\sum_{i=0}^{L} N/(Mn)(m/n)^i) = O(nN/(M(1 - m/n)))$. Note at each level i in we shuffle at most $N(2m/n)^i$ objects.

In the TD phase, each Mapper task reads the P top pairs as side input and a stream of objects. For each object, it emits 0 up to P values; however in total, the Map phase cannot emit more than $2Pn$ objects to the Reduce phase. These are shuffled to P Reducer tasks. The Mappers also emit $O((nL_{i-1} - P)k)$ near neighbor pairs for objects whose representatives were not matched to the DistributedTopP computation. Each Reducer task receives at most $2n$ clustered objects and performs $O(n^2)$ distance computations. It updates the near neighbors in the objects that matched in $O(n^2 + nk)$ time. Finally, it writes these clustered objects in $O(n)$ time and then emits the $O(n^2)$ newly computed related pairs. With M machines, the total Reduce phase running time in each level is then $O(P(n^2 + nk)/M)$. Finally, during the corresponding DistributedTopP computation, each task reads $O(p)$ near neighbors and returns the top P in time linear in p. Hence, each MapReduce iteration in this computation takes time $O(E/(pM))$ (where E is defined above).

Conceptually, the Merge phase consists of a single MapReduce where each reducer task processes all the instances of an object (written at different levels in BU and TD) and computes the final consolidated k near neighbors for the object. The total number of objects produced in the BU phase is $\sum_{i=0}^{L} N(m/n)^i$; whereas the total produced in the TD phase is bounded by nPL. Hence, the number of objects that are shuffled in this phase is $nPL + N/(1 - m/n)$. Note that in the worst case, the TD phase will produce up to P instances for the same object in each of the L levels, resulting in a running time of $O(kPL)$ for the reducer task with that input. To ameliorate this for such objects, we can perform a distributed computation to compute the top k.

B. Quality Analysis

Proving error bounds on the quality of the near neighbors identified by SAJ w.r.t. optimal nearest neighbors turns out to be a difficult theoretical challenge. For this paper, we provide the following Lemma as an initial step and leave the rest of the theoretical work for the future. In Section VI, we provide extensive quality analysis through empirical studies.

Lemma 1: Assume m, the number of clusters in the BU phase, is chosen appropriately for the data. For any two objects a and b, if a's neighbors are closer to a than b's neighbors are to b, then SAJ (specifically the TD phase) favors finding a's neighbors as long as the number of levels in the BU phase is at least 3. Further, the larger P is, the more of b's neighbors will be updated in the TD phase.

Proof Sketch. The theorem is essentially a consequence of the inter-partition ranking of distances between compared pairs at each level in the TD phase, and the cut-off rank P.

A good clustering is needed so that dense sets of objects have a cluster representative at the end of the BU phase. Since SAJ's BU phase is based on [34], we know that the set of representatives at the top is near optimal (note that we stop the BU computation when the representatives fit in a machine's memory, however this set will contain the final set of cluster representatives). The minimum number of levels is necessary since the representatives of dense sets of objects may be far from all other representatives at the end of the BU phase (they will also have fewer representatives), hence they may not be chosen in the initial TopP step. However, such pairs will participate in subsequent levels since they are output. The pairs compute with all other pairs at the same level to be included in the following TopP. This per-level global ranking causes far pairs to drop from consideration. The larger the P, the more chance these pairs have to be included in the TopP. □

For example, suppose the data contains two well-separated sets of points, R_1 and R_2 where R_1 is dense and points in R_2 are loosely related. Also, let $m = 4$ and $P = 1$. The BU phase will likely end up with one representative from R_1, call it r_1, and 3 from R_2, r_{21}, r_{22}, and r_{23}. Since R_1 and R_2 are well separated, r_1 will not participate in the pair chosen at the top level. However, (r, r_1) for $r \in R_1$ will be considered and since the points are dense, all pairs from R_2 will drop from consideration, and the neighbors computed for R_2 will thus not be updated in the TD phase.

A refinement to SAJ is suggested by this theorem. Given the output of the algorithm, objects with far neighbors may have nearer neighbors in the data set, however they were not compared with these neighbors as the pairs did not make the distance cut-off P. Consider the $N/2$ objects with the relatively nearest neighbors (here, the factor can be chosen as desired), and remove them from the set. Now run SAJ on the remaining objects. The newly computed object neighbors from the second run must be merged with their neighbors that were computed in the first run. These refinements can be continued for the next $N/4$ objects and so on as needed. A complementary minor refinement would be to compute a TopP+ list at each level that reserves a small allowance for large distances to be propagated down.

VI. Experimental Evaluation

In this section, we evaluate SAJ via comprehensive experiments on both scalability (Section VI-B) and quality (Section VI-C). We begin by describing the experimental setup.

A. Experimental Setup

Datasets: We adopt three datasets for our experiments, *DBLP*, *PointsOfInterest*, and *WebTables*.

- DBLP[5]: This dataset consists of citation records for about 1.7 million articles. Similar to other studies that have used this dataset, scale it up to $40\times$ (68 *million* articles) for scalability testing by making copies of the entries and inserting unique tokens to make them distinct. We adopt the same join function as in [6], which is Jaccard Similarity over *title* and *author* tokens, but without using the similarity threshold.
- PointsOfInterest: This dataset consists of points of interests whose profile contains both geographical information and content information. They are modeled after Places[6]. We generate up to 1 *billion* objects and use a function that combines great-circle distance and content similarity.
- WebTables: This real dataset consists of 130 *million* HTML tables that is a subset of the dataset we use in our Table Search[7] project [3]. In particular, objects within this dataset are often very large due to auxiliary annotations, resulting in on-disk size of 2TB (*after compression*), which is at least $200\times$ the next largest dataset we have seen in the literature. The join function for this dataset is a complex function involving machine learning style processing of the full objects.

With the exception of the join function for DBLP, the join functions for both PointsOfInterest and WebTables are impossible to analyze because they are not in L_p space or based on set similarities. In fact, our desire to join the WebTables dataset is what motivated this study.

Baselines: Despite many recent studies [10], [7], [11], [5], [6], [14] on the large-scale join problem, none of these is comparable to SAJ due to our unique requirements to support complex join functions, the ability to handle objects not representable as sets or multi-dimensional points, and to perform on extremely large datasets. Nonetheless, to put the scalability of SAJ in perspective and understand it better, we compare our system with FuzzyJoin [6], the state of the art algorithm for set similarity joins based on MapReduce. We choose FuzzyJoin, instead of a few more recent multi-dimensional space based techniques (e.g., [7]), because the DBLP dataset naturally lends itself to set-similarity joins, which can be handled by FuzzyJoin, but not by those latter works. We only perform the quality comparison for DBLP because neither set similarity nor L_p distance are applicable for the other two datasets.

System Parameters: All experiments were run on Google's MapReduce system with the number of machines in the range of tens ($1\times$) to hundreds ($18\times$), with $18\times$ being the default. (Due to company policy, we are not able to disclose specific numbers about the machines.) To provide a fair comparison, we faithfully ported FuzzyJoin to Google's MapReduce system. We analyze the behavior of SAJ along the following main parameter dimensions: *N*, the total number of objects; k, the number of desired neighbors per object; P/N, the ratio between the number of top pairs P and N; and the (relative) number of machines used. Table III summarizes the default values for parameters as described in Table I, except for N, which varies for each dataset.

B. Scalability

In this section we analyze the scalability of SAJ. In particular, we use the DBLP dataset for comparison with FuzzyJoin,

[5]http://dblp.uni-trier.de/xml/

[6]http://places.google.com/

[7]http://research.google.com/tables

Parameter	n	m	p	N	k	P/N
Default Values	10^3	10	10^8	varies	20	1%

TABLE III
DEFAULT PARAMETER VALUES.

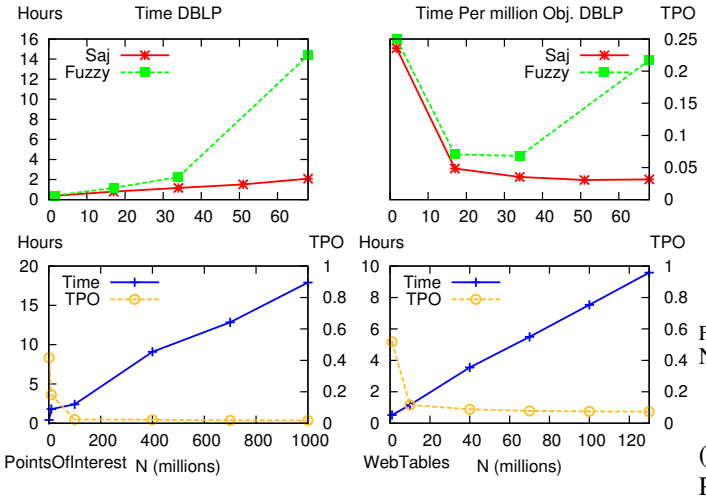

Fig. 3. Scalability Analysis Varying N: (top) Comparison with FuzzyJoin over DBLP; (bottom) PointsOfInterest and WebTables. ($k = 20, P/N = 1\%$)

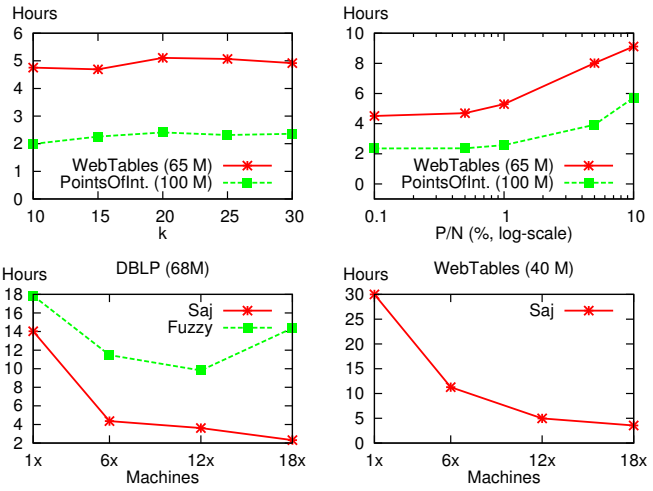

Fig. 4. Scalability Analysis Varying k (top-left), P (top-right), and Number of Machines (bottom).

and the PointsOfInterest and WebTables datasets for super large scalability analyses where the former has a skewed distribution and the latter has very large objects. The parameters we vary are N, k, P, and machines, but not p since it is determined by machine configuration.

Varying N. Figure 3 (top) compares the running time of SAJ and FuzzyJoin (with threshold value 0.8 as in [6]) over the DBLP dataset with N varying from 1.7 million to 68 million. The left panel shows SAJ scales linearly with increasing N and performs much better than FuzzyJoin especially when the dataset becomes larger. The right panel analyzes the time spent per object, which demonstrates the same trend. The results are expected. As the dataset becomes larger, data skew becomes increasingly common and leads to more objects appearing in the same candidate group, leading to straggling reducers that slows down the entire FuzzyJoin computation. Meanwhile, SAJ is designed to avoid skew-caused straggling reducers and therefore scales much more gracefully. Note that a disk-based solution was introduced in [6] to handle large candidate groups. In our experiments, we take it further by dropping objects when the group becomes too large to handle, so the running time in Figure 3 for FuzzyJoin is in fact a lower bound on the real cost.

Figure 3 (bottom) illustrates the same scalability analyses over the PointsOfInterest and WebTables datasets. FuzzyJoin does not apply for those two datasets due to its inability to handle black-box join functions. Despite the different characteristics of these two datasets—PointsOfInterest with a skewed distribution and WebTables with very large objects—SAJ is able to scale linearly for both. We note that the running time (overall and per-object) for WebTables is longer than that for PointsOfInterest with the same number of objects. This is not surprising because the objects in WebTables are much larger and the join function much more costly. For all three datasets we notice that time-spent per object is fairly large when the dataset is small. This is not surprising since the cost of starting MapReduce jobs makes the framework ill-suited for small datasets.

Varying k and P/N. Next, we vary the two parameters k and P and analyze the performance of SAJ over DBLP and WebTables datasets. (The results for the PointsOfInterest dataset are similar.) Figure 4 (top-left) shows that increasing k up to 30 does not noticeably affect the running time of SAJ. While surprising at first, this is because only a small identifier is required to keep track of each neighbor and the cost of maintaining the small list of neighbors is dwarfed by the cost of comparing objects using the join functions.

Figure 4 (top-right) shows that the running time of SAJ increases as the ratio of P/N increases. Parameter P controls the number of pairs retained in the TD phase to guide the comparisons in the next iteration. The higher the P, the more comparisons and the better the result qualities are (see Section VI-C for caveats). The results here demonstrate one big advantage of SAJ: the user can control the tradeoff between running time and desired quality by adjusting P accordingly. In practice, we have found that choosing P as 1% of N often achieves a good balance between running time and final result quality.

Varying Number of Machines. Finally, Figure 4 (bottom) shows how SAJ performs as the available resources (i.e., number of machines) change. We show four data points for each set of experiments, where the number of machines range from $1\times$ to $18\times$. For both DBLP and WebTables datasets, SAJ is able to take advantage of the increasing parallelism and reduce the running time accordingly. As expected, the reduction slows down as parallelism increases because of

the increased shuffling cost among the machines. In contrast, while FuzzyJoin runs faster as more machines are added initially, it surprisingly slows down when more machines are further added. Our investigation shows that, as more and more machines are added, the join function-based pruning is applied to smaller initial groups leading to reduced pruning power, hence the negative impact on the performance.

C. Quality

To achieve the super scalability as shown in Section VI-B, SAJ adopts a best-effort approach that produces near neighbors. In this section, we analyze the quality of the results and demonstrate the effectiveness of the TD phase, which is the key contributor to both scalability and quality. Since identifying the near*est* neighbors is explicitly a non-goal, we do not compute the percentage of nearest neighbors we produce and use **Average-Distance** to measure the quality instead of precision/recall. The *Average-Distance* is defined as the average distance from the produced near neighbors to the objects: i.e., $\text{Avg}_{i \in R}(\text{Avg}_{j \in S_i} F_J(i,j))$, where each j is a produced neighbor of object i and F_J is the join function.

The tunable parameters are k and P (m is not tunable because it is determined by the object size in the dataset). We perform all quality experiments varying these parameters over the three datasets with $N = 10^6$ objects, and analyze the results using 1000 randomly sampled objects. We compare the following three approaches. First, we analyze near neighbors produced purely by the BU phase, this is equivalent to a random partition approach and serves as the **BU-Baseline**. This measurement also represents the expected distance of two randomly selected objects. Second, we analyze the near neighbors that SAJ produces for objects that participates in at least one comparison during the TD phase, and we call this **TD-Updated**. Third, we analyze *Nearest*, which are the actual nearest neighbors of the sampled objects.

Varying k. The left panel of Figure 5 shows that, for all datasets, TD-Updated significantly improves upon BU-Baseline by reducing the average distances. We also observe that the distances of our near neighbors to the nearest neighbors are much smaller than those of the baseline neighbors to the nearest neighbors. This improvement demonstrates that the TD phase has a significant positive impact on quality and guides the comparisons intelligently. In particular, for the PointsOfInterest dataset, even though the dataset is very dense (i.e., all the objects are very close to each other), which means even the baseline can generate good near neighbors (as shown by the small average distances in the plot), SAJ is still able to reduce the distance by 50%. For the WebTables dataset, SAJ performs especially well due to the fact that objects in this dataset often have a clear set of near neighbors while being far away from the rest of the objects. Finally, as k goes up, the quality degrades as expected because larger k requires more comparisons.

Varying P. The right panel of Figure 5 shows the impact of the TD phase when varying P. Similar to the previous

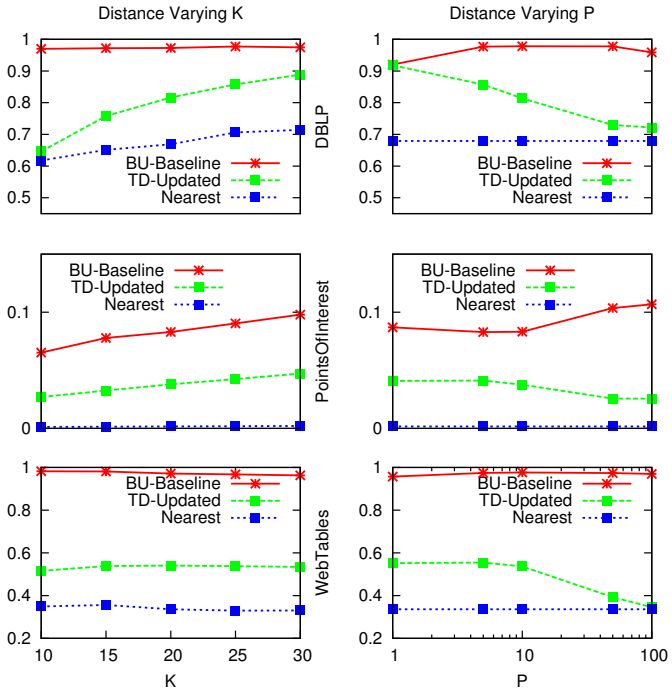

Fig. 5. Average-Distance for DBLP (top), PointsOfInterest (middle), and WebTables (bottom), varying k (left, $P = 10000$) and P (right, $k = 20$) (000s), $N = 10^6$.

experiments, for almost all values of P, TD phase significantly improves the quality over the baseline BU-Baseline for all three datasets even when P is set to a very small number of 1000 (except for DBLP). As expected, the quality improves as P increases because more comparisons are being performed. While the quality continues to improve as P approaches 10% of N (the total number of objects in the dataset) it is already significantly better than the baseline when P is 1%. $P = N/100$ is a reasonable value for most practical datasets and it is indeed the number we use in our scalability experiments in Section VI-B.

Precision. Finally, we again emphasize that having near*est* neighbors is not crucial in our target application of finding related tables on the Web. We may fail to locate any of the most relevant tables, but the users can still be happy with the tables we return because they are quite relevant and much better than random tables. However, for the sake of completeness, we also measured the percentage of nearest neighbors produced by our system for the WebTables dataset, as shown in Figure 6. We observe that we find a good percentage of nearest neighbors even though this is not the goal of SAJ. The number of nearest neighbors grows as P increases, approaching gracefully the exact solution. The impact of the TD phase is again significant.

We also measured the percentage of nearest neighbors for the DBLP dataset using the default settings. The percentage of objects that contain at least one of the true Top-2 in their neighbor list produced by SAJ is 17.8% and the percentage of

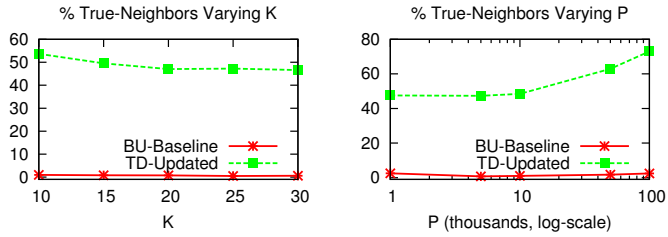

Fig. 6. True-Neighbors (%) for WebTables, varying k (left, $P = 10000$) and P (right, $k = 20$) (000s), $N = 10^6$.

objects that contain at least one of the true Top-5 is 50.9%.

VII. Conclusion & Future Work

We presented the SAJ system for super scalable join with complex functions. To the best of our knowledge, this is the first scalable join system to allow complex join functions, and we accomplish this by employing the best-effort *near* neighbor join approach. The keys to our scalability are two-fold. First, unlike previous parallel approaches, SAJ strictly adheres to the machine task capacity and carefully avoids any possible bottlenecks in the system. Second, through the parameter P, SAJ allows the user to easily trade-off result quality with resources availability (i.e., number of machines) by tuning the result quality requirements. Extensive scalability experiments demonstrate that SAJ can process real world large scale datasets with billions of objects on a daily basis. Quality experiments show that SAJ achieves good result quality despite not having the knowledge of the join function.

References

[1] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *SIGMOD*, 2010.

[2] J. Dean and M. R. Henzinger, "Finding related pages in the World Wide Web," *Computer Networks*, vol. 31, no. 11-16, pp. 1467–1479, 1999.

[3] C. Yu, "Towards a high quality and web-scalable table search engine," in *KEYS*, 2012, p. 1.

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[5] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *SIGMOD*, 2011.

[6] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *SIGMOD*, 2010.

[7] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," *PVLDB*, vol. 5, no. 10, 2012.

[8] R. Baraglia, G. D. F. Morales, and C. Lucchese, "Document similarity self-join with MapReduce," in *ICDM*, 2010.

[9] T. Elsayed *et al.*, "Pairwise document similarity in large collections with MapReduce," in *ACL*, 2008.

[10] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using MapReduce," in *ICDE*, 2012.

[11] A. Metwally and C. Faloutsos, "V-SMART-Join: A scalable MapReduce framework for all-pair similarity joins of multisets and vectors," *PVLDB*, vol. 5, no. 8, 2012.

[12] J. C. Shafer and R. Agrawal, "Parallel algorithms for high-dimensional similarity joins for data mining applications," in *VLDB*, 1997.

[13] S. Voulgaris and M. van Steen, "Epidemic-style management of semantic overlays for content-based searching," in *Eruo-Par*, 2005.

[14] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *EDBT*, 2012.

[15] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006.

[16] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007.

[17] Y. Chen and J. M. Patel, "Efficient evaluation of all-nearest-neighbor queries," in *ICDE*, 2007.

[18] N. Koudas and K. C. Sevcik, "High dimensional similarity joins: Algorithms and performance evaluation," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 1, pp. 3–18, 2000.

[19] S. Chaudhuri *et al.*, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006.

[20] C. Xia, H. Lu, B. C. Ooi, and J. Hu, "Gorder: An efficient method for kNN join processing," in *VLDB*, 2004.

[21] C. Xiao, W. Wang, and X. Lin, "Ed-Join: an efficient algorithm for similarity joins with edit distance constraints," *PVLDB*, vol. 1, no. 1, pp. 933–944, 2008.

[22] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008.

[23] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *ICDE*, 2009.

[24] C. Böhm and F. Krebs, "High performance data mining using the nearest neighbor join," in *ICDM*, 2002, pp. 43–50.

[25] B. Yao, F. Li, and P. Kumar, "K nearest neighbor queries and knn-joins in large relational databases (almost) for free," in *ICDE*, 2010, pp. 4–15.

[26] E. H. Jacox and H. Samet, "Metric space similarity joins," *ACM Trans. Database Syst.*, vol. 33, no. 2, 2008.

[27] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.

[28] H. Lee, R. T. Ng, and K. Shim, "Similarity join size estimation using locality sensitive hashing," *PVLDB*, vol. 4, no. 6, pp. 338–349, 2011.

[29] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *GeoInformatica*, vol. 14, no. 1, pp. 55–82, 2010.

[30] J. Wang, S. Kumar, and S.-F. Chang, "Sequential projection learning for hashing with compact codes," in *ICML*, 2010.

[31] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b^+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.

[32] T. cker Chiueh, "Content-based image indexing," in *VLDB*, 1994, pp. 582–593.

[33] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 426–435.

[34] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams," in *FOCS*, 2000.

[35] Apache, ""apache hadoop, http://hadoop.apache.org/"." [Online]. Available: http://hadoop.apache.org/

[36] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[37] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[38] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," *SIGPLAN Not.*, vol. 45, pp. 363–375, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806638

[39] A. Lukasová, "Hierarchical agglomerative clustering procedure," *Pattern Recognition*, vol. 11, no. 5-6, pp. 365–381, 1979.