HEADS-JOIN: Efficient Earth Mover's Distance Similarity Joins on Hadoop

Jin Huang, Rui Zhang, *Member, IEEE*, Rajkumar Buyya, *Fellow, IEEE*,
Jian Chen *Member, IEEE*, Yongwei Wu, *Member, IEEE*

Abstract—The Earth Mover's Distance (EMD) similarity join has a number of important applications such as near duplicate image retrieval and distributed based pattern analysis. However, the computational cost of EMD is super cubic and consequently the EMD similarity join operation is prohibitive for datasets of even medium size. We propose to employ the Hadoop platform to speed up the operation. Simply porting the state-of-the-art metric distance similarity join algorithms to Hadoop results in inefficiency because they involve excessive distance computations and are vulnerable to skewed data distributions. We propose a novel framework, named HEADS-JOIN, which transforms data into the space of EMD lower bounds and performs pruning and partitioning at a low cost because computing these EMD lower bounds has constant or linear complexity. We investigate both range and top-k joins, and design efficient algorithms on three popular Hadoop computation paradigms, i.e., MapReduce, Bulk Synchronous Parallel, and Spark. We conduct extensive experiments on both real and synthetic datasets. The results show that HEADS-JOIN outperforms the state-of-the-art metric similarity join technique, i.e., Quickjoin, by up to an order of magnitude and scales out well.

Index Terms—Earth Mover's Distance, Similarity Join, MapReduce, Bulk Synchronous Parallel

1 INTRODUCTION

The similarity join retrieves all the pairs of objects from two datasets such that the similarity between the two objects in every pair is high. The similarity measure employed in the join predicate determines which data objects are similar to each other and thus has a major influence on the effectiveness of the join operation. The *Earth Mover's Distance* (EMD) is an attractive measure for applications such as content-based image retrieval [21], video-based gesture recognition [18], and near duplicate detection [26]. Two examples are as follows.

Example 1. *Given a set of copyright images and a set of user-uploaded images, where images are represented as color distributions, we can use similarity joins to detect potential copyright infringement among the user-uploaded contents. Here, the EMD is a better similarity measure than the ℓ_p distances. This is because that the EMD can capture the facts such as the orange dimensions are similar to the yellow and the red dimensions; for ℓ_p distances, all dimensions are independent and indistinguishable from each other.*

Example 2. *Given a set of mobile usage patterns, which are represented as distributions of hourly usage counts (in a 24d space), similarity joins can be used to figure out the similar users [2]. In this task, the EMD is better than ℓ_p because it captures the underlying relations between the timestamps (which are dimensions). For example, a call on 9am is similar to a call on 10am (a morning working call) rather than a call on 9pm (an evening family call).*

Generally, the EMD incorporates the underlying correlations between different dimensions in addition to the difference on the same dimensions. For applications where such correlations matter to the effectiveness, the EMD is a more proper choice than distances that ignore the correlations, e.g., the ℓ_p distances.

Specifically, the EMD defines the dissimilarity as the *minimal transformation cost* between two data objects. However, this minimization is a transshipment problem which has the complexity of $O(n^3 \log n)$. In our experiment on a machine with 2.8GHz CPU, a single EMD computation on two histograms with 32 two-dimensional bins ($n = 32$) consumes 50 ms, which is about 25,000 times of the ℓ_2 distance's 0.002 ms on the same histograms. In real applications, datasets may contain hundreds of thousands or even millions of objects. An EMD similarity join on them may *take weeks to months to complete on a single machine*. To enable effective and efficient EMD analysis in practice, leveraging a cluster of shared nothing machine is a promising approach. This is especially true as the emerging cloud computing techniques have made such clusters both highly available and unprecedentedly economical.

We propose to use the popular platform for a distributed cluster, Hadoop, to tackle this problem. We design our solutions based on the MapReduce [9] (MR), the Bulk Synchronous Parallel [23] (BSP), and the Spark [28] paradigms. All these paradigms can run upon a Hadoop

- *Jin Huang, Rui Zhang, and Rajkumar Buyya are with the Department of Information and Computing Systems, University of Melbourne, Melbourne, VIC 3010, Australia. E-mail: {huang.j, rui.zhang, rbuyya}@unimelb.edu.au*
- *Jian Chen (the corresponding author) is with the School of Software Engineering, South China University of Technology, Guangzhou, China. E-mail: ellachen@scut.edu.cn*
- *Yongwei Wu is with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China. E-mail: wuyw@tsinghua.edu.cn*

cluster. There have been enormous efforts in implementing join operations using MR [14], [16], [17], [22]. However, the state-of-the-art technique designed for metric space similarity joins, i.e., *Quickjoin* [13], is inefficient for EMD similarity joins in the parallel environment due to the below challenges.

- **Challenge 1**: Existing techniques such as Quickjoin require performing pruning and partitioning in the EMD space, which is prohibitive.
- **Challenge 2**: Real-life data is usually skewed. Quickjoin may have highly unbalanced workloads in case of skewed data, which results in long completion time.

To overcome Challenge 1, we propose a novel framework named *Hadoop EArth mover's Distance Similarity Join* (HEADS-JOIN[1]). Instead of pruning dissimilar pairs and partitioning data objects in the EMD space, HEADS-JOIN transforms data objects into the space of EMD lower bounds, where the pruning and partitioning can be performed based on the EMD lower bounds; these lower bounds can be computed at a low cost in the transformed low-dimensional space, which is much cheaper than computing EMD in the original high-dimensional space. Conceptually, HEADS-JOIN has three phases.

1) Transform data objects into spaces corresponding to multiple normal lower bounds of EMD.
2) Divide the transformed spaces using a set of specially designed grids based on the characteristics of the lower bounds and group the transformed records into composite cells.
3) Compute the lower bound of the EMD between every record and every cell; any $\langle record, cell\rangle$ pair that has a lower bound of EMD greater than the threshold is pruned. The remaining $\langle record, cell\rangle$ pairs will be partitioned and go through further refinement steps.

To overcome Challenge 2, we propose to use a quantile based grid technique and a cardinality based grouping technique to balance the workloads of the refinement steps in the third phase of HEADS-JOIN.

This paper extends our preliminary work [12], where we proposed a MR based framework for EMD similarity *range* joins. In this paper, we make the following *extra contribution*.

- We further enhance the framework to incorporate the upper bound and the centroid lower bound of EMD (Section 3.3).
- We extend the framework proposed in the previous work to handle top-k joins in addition to range joins (Section 3.4).
- Observing that MR has overhead due to the lack of synchronization within a job, we extend the framework by designing algorithms for the BSP (Section 4) and the Spark (Section 5) paradigms. The extended framework is named HEADS-JOIN.
- We re-implement all the algorithms using the latest version of Apache Hadoop MapReduce, Apache Hama, and Apache Spark. We conduct brand new experiments on both real and synthetic datasets and evaluate our proposed algorithms against the state-of-the-art techniques in terms of both time and communication cost (Section 7) .

The remainder of the paper is organized as follows. Section 2 presents the preliminaries. Section 3, Section 4, and Section 5 elaborate the algorithms designed for MR, BSP, and Spark, respectively. Section 6 discusses related studies. Section 7 gives the empirical results and Section 8 concludes the paper.

2 BACKGROUND

We first present problem formulation and then briefly describe the techniques that HEADS-JOIN is built on: the EMD bounding techniques and the Hadoop computation paradigms.

2.1 Earth Mover's Distance and Problem Definition

In our problem, data objects are represented as histograms and in the remainder of the paper *we refer to the histogram of a data object simply as a histogram*. A histogram h is represented as n bins. Each bin consists of a location which is a multi-dimensional vector $\vec{l_i}$ and a weight which is a nonnegative value w_i. The EMD between a pair of histograms is the *minimum cost* of transforming one histogram to the other histogram, where the cost is defined as the amount of weight moved times the *ground distance* between bins that the weight is moved [19]. We follow the literature [20], [25], [27] and focus on EMD with the ℓ_2 ground distance in this paper. Formally,

$$EMD(h_\alpha, h_\beta) = \min \sum_i^n \sum_j^n f_{i,j} d_{i,j}$$

$$s.t. \forall i : \sum_j f_{i,j} = w_i;\ \forall j : \sum_i f_{i,j} = w_j;\ \forall i,j : f_{i,j} \geq 0,$$

where $d_{i,j}$ is the ground distance between the i^{th} and the j^{th} bins, i.e., $d_{i,j} = d_{\ell_2}(\vec{l_i}, \vec{l_j})$. Computing EMD is solving a special case of the Kantorovich-Rubinstein transshipment problem, where the transportation simplex method provides the average-case time complexity of $O(n^3 \log n)$.

We consider two variants of EMD similarity joins, i.e., ***range*** and ***top-k*** joins. Range joins retrieve all the pairs with EMD smaller than a given threshold ϵ. Top-k joins retrieve the k pairs that have the smallest EMD.

2.2 Bounds of EMD Employed in HEADS-JOIN

Bounds of EMD are useful for similarity joins since we can avoid excessive cost in computing EMD by filtering dissimilar pairs with lower bounds and identifying similar pairs with upper bounds. Below we briefly describe the bounds used in HEADS-JOIN; more details can be found in [12].

1. The 'Hadoop' here indicates both Hadoop and Hadoop-alike systems, e.g., Spark, as long as they are compatible to HDFS and YARN

2.2.1 Normal Lower Bound

To compute the normal lower bound (normal-LB) between a histogram and a group of histograms, there are five steps [20].

1) ***Projecting*** *the original histograms to one-dimensional histograms* by timing the locations in the original histogram to a random projection vector.
2) *Constructing Cumulative Distribution Functions (**CDF**)* for the projected histogram.
3) *Approximating the histogram CDF with a **normal** distribution*. The μ and σ for the normal CDF can be easily computed by fitting the normal CDF to the histogram CDF and the approximation error values C are obtained by comparing the histogram CDF with the normal CDF.
4) ***Transforming*** *normal CDF to Hough normal space*. For a normal CDF $\Phi(\mu, \sigma^2)$, we transform it to a record (m, b), where $m = \frac{1}{\sigma}$ and $b = \frac{-\mu}{\sigma}$. *Each transformed record in the Hough normal space therefore corresponds to one histogram in the original space*.
5) ***Grouping*** *transformed records into diamond-shape region and computing the normal-LB*. The transformed records are grouped to a diamond-shape region using lines with slopes of $-t_{min}$ and $-t_{max}$. For a record h and a region g, let h' denote the the projection of h on its nearest edge of g, t denote the intersection between Φ_a and Φ_b, we have

$$EMD(h, h_g \in g) \geq LB_{normal}(h, g)$$
$$= \begin{cases} N(h, g_N) + E(h, g_N) & \text{if } h \text{ dominates } g \\ F(h, h', \arg\min_{g=\{g_N, g_S\}} d_{\ell_2}(g, h)) & \\ \qquad \text{if } h \text{ partially dominates } g & \\ \min(F(h, g_R, g_N), F(h, g_R, g_S)) & \\ \text{if } h \text{ has no domination relationship with } g & \\ N(h, g_S) + E(h, g_S) \text{ if } h \text{ is dominated by } g \end{cases}$$

where,

$$F(a, b, c) = \frac{1}{2}(N(a, c) + N(b, a) - N(b, c)) + E(a, c),$$
$$N(a, b) = \begin{cases} |\int_{t_{min}}^{t} \Phi_a - \int_{t_{min}}^{t} \Phi_b| + \\ |\int_{t}^{t_{max}} \Phi_a - \int_{t}^{t_{max}} \Phi_b| \\ \qquad \text{if } t_{min} \leq t \leq t_{max} \\ |\int_{t_{min}}^{t_{max}} \Phi_a - \int_{t_{min}}^{t_{max}} \Phi_b| \\ \text{if } t_{min} > t \text{ or } t > t_{max}, \end{cases}$$
$$E(a, b) = \begin{cases} C_a[t] - C_b[t] & \text{if } \Phi_b \text{ dominates } \Phi_a \\ C_b[t] - C_a[t] & \text{if } \Phi_a \text{ dominates } \Phi_b \end{cases}$$
$$g_R = \arg\min_{g=\{g_W, g_E\}} d_{\ell_2}(g, h),$$

where g_N, g_W, g_S, and g_E represent the northern, western, southern, and eastern vertex of g, respectively. The dominance relationship between h and g is determined by the stochastic dominance between their corresponding normal CDF.

The normal-LB can be computed at an $O(1)$ cost since i) two normal CDF with different σ only have one intersection; ii) there is a closed-form formula on integrating normal CDF, i.e., $\int_{x_1}^{x_2} \Phi(\mu, \sigma^2) = \sigma(x_2\Phi_{std}(x_2) + \phi_{std}(x_2) - x_1\Phi_{std}(x_1) - \phi_{std}(x_1))$, where Φ_{std} and ϕ_{std} are the CDF and density function of the standard normal distribution $\mathcal{N}(0, 1)$, respectively.

2.2.2 Dual Lower Bound

The dual lower bound (dual-LB) [27] is computed via *feasible solutions* to the dual form problem of the optimization in EMD. A feasible solution, denoted by Π, is a set of variables satisfying the constraints of its dual form problem, with a *solution key*, denoted by π, computed by aggregating these variables. The feasible solution can be computed by sampling the histograms. Each feasible solution can transform a histogram to two one-dimensional values λ and ρ, denoted as *dual keys*. Given two histograms h and h', two lower bounds can be computed as $(\lambda_h + \rho_{h'})$ and $(\lambda_h - \lambda_{h'} + \pi)$, respectively.

2.2.3 Centroid Lower Bound

Given that an ℓ_p distance is used as the ground distance, the centroid lower bound (centroid-LB) [19] is of linear cost and is computed as the ground distance between the centroids of the two equal-weight histograms. As we consider ℓ_2 ground distance, the centroid-LB is $d_{\ell_2}(\sum_i w_i l_i, \sum_j w_j l_j)$, where we coin $\omega_h = \sum_i w_i l_i$ as the *centroid key*.

2.2.4 Reference Upper Bound

The reference upper bound (reference-UB) uses several random reference histogram $\hat{h}$ and the triangle inequality of metric distances to bound EMD between two histograms. Since EMD is the minimum flow between two histograms, we can compute a *feasible flow* between two histograms as its natural upper bound. Formally,

$$\begin{aligned} UB_{\hat{h}}(h_\alpha, h_\beta) &= F(h_\alpha, \hat{h}) + F(h_\beta, \hat{h}) \\ &\geq EMD(h_\alpha, \hat{h}) + EMD(h_\beta, \hat{h}) \\ &\geq EMD(h_\alpha, h_\beta), \end{aligned}$$

where the feasible flow $F(h, \hat{h})$ can be easily computed by first sorting the bins of the two histograms separately on their weights, and then constructing flow from one to the other following the descending order of weights in a greedy manner.

2.3 Parallel Computation Paradigms in Hadoop

Hadoop[2] is a massively popular distributed data processing platform. It runs a cluster of commodity machines and assumes the CPUs as well as the main memories are isolated on distributed machines while a distributed file system (HDFS in this case) is available. There are a number of different computation paradigms

2. Apache Hadoop, `http://hadoop.apache.com`

available over the platform, among which MR, BSP, and Spark have attracted a large number of users.

The MR paradigm [9] is designed to simplify parallel batch processing on a cluster of commodity machines. It involves a `map` and a `reduce` function. The `map` and `reduce` functions are executed independently on the machines referred to as the mappers and the reducers. Briefly, the mappers process key-value pairs and shuffle intermediate pairs to reducers. MR guarantees that intermediate pairs with the same key value are shuffled to the same reducer. The reducers then aggregate values based on the received pairs. A `map-reduce` procedure corresponds to a *MR job*.

The BSP paradigm [23] is a generic computation paradigm designed for parallel computation units. In BSP, computations are executed on parallel workers in the fashion of consecutive *supersteps*. In each superstep, each worker first computes independently using the local data, then *sends messages* to each other, and *reads messages* sent to them in a *synchronous barrier*.

The Resilient Distributed Dataset (RDD) [28] consists of a fault-tolerant distributed dataset and a set of computation interfaces that can be operated on this dataset. The dataset is stored by leveraging the available memory hierarchy (e.g., smart caching [30], [1]), and the interfaces are inspired by the functional programming concepts, e.g., `map`, `reduce`, `filter`, `fold`, etc. Apache Spark[3] is the open source implementation of RDDs, and it is fully compatible to the Hadoop platform. In the remainder of this paper, we use Spark and RDD interchangeably.

3 HEADS-JOIN ON MAPREDUCE

HEADS-JOIN employs the normal-LB to transform data objects into the Hough normal space, divides the space into grids, collects statistics on grouped records in the cells, and then conducts pruning, partitioning, and refining. With MR, this procedure is implemented using three MR jobs. Below we briefly describe these jobs for the range joins; for more details of each MR job involved, please refer to our preliminary study [12]. At the end of this section, we elaborate how HEADS- JOIN handle top-k joins with some moderate modifications.

3.1 Job 1: Obtaining the Domain of Transformation

We apply p projection vectors, and therefore use p different normal-LB in the later pruning. This job is designed for obtaining the domains of the p transformed Hough normal spaces. For each projection vector v, on the mappers, the normal-LB technique is applied to transform each histogram to multiple 2d records. One reducer aggregates quantile values of each dimension of a space, serving as i) the domain and ii) the base of grid division.

We use the quantile values computed by the reducers to conduct the grid division. These quantile values Q are distributed to workers for further use by the following job. Then the space can be divided into z^2 grid cells. As described in Section 2.2, lines with slopes of $-t_{max}$ and $-t_{min}$ are used to conduct the division. In order to cover the space domain, the grid is a *minimum bounding quadrilateral* of the domain.

Lastly, for lower bounds that require global information, e.g., the feasible solutions in the dual-LB, we can use the `cleanup` phase to compute that information. For the reference-UB, a small number of reference histograms are sampled and distributed to workers for further accesses.

3.2 Job 2: Computing the Approximation Errors

To prune dissimilar composite cells for a histogram, we need to compute the normal-LB between them, which requires the aggregated approximation error values for the composite cells. We employ another MR job to aggregate these values. In addition to the approximation errors, we are also interested in two types of information. First, we collect the cardinality of records in each composite cell, which will be helpful for conducting load balancing in the following job. Second, as we will employ both the dual-LB and the centroid-LB to prune the potential pairs, the lower bound key ranges for these two lower bounds are also aggregated for each composite cell. The aggregation is implemented as a simple job, which applies the bounding techniques on mappers, uses the composite cell as the ID, and aggregates all necessary information on reducers.

3.3 Job 3: Pruning, Partitioning, and Refining

In this job, a mapper prunes the dissimilar composite cells for each histogram and distributes the histogram to different reducers based on the pruning results. The reducers further prune candidate pairs using a chain of lower bounds.

We use the centroid-LB as an example of plugging lower bounds into pruning. The dual-LB is applied in a similar way. The key range of centroid-LB is multi-dimensional, i.e., same to the dimension of the bin locations. Hence, the lower bound between a centroid key and a range is the *minimum distance* between a multi-dimensional point ω_h, and a multi-dimensional orthotope defined by ω_{min} and ω_{max}. A composite cell G with $[\omega_{min}, \omega_{max}]$ is pruned for histogram h if

$$LB_{centroid} = min\{d(\omega_h, \omega_G), \omega_G \in [\omega_{min}, \omega_{max}]\} > \epsilon.$$

Before distributing records to different reducers based on the composite cells, we try to organize the composite cells such that reducers are of similar amount of workloads. We propose to group composite cells based on the number of records in them and assign grouped composite cells to reducers accordingly. This can be done by a simple algorithm. Given the number of groups, first we sort the composite cell in a descending order on the

3. Apache Spark, `http://spark.apache.com`

number of records in it, which is actually the byproduct of Job 2. Then we assign composite cells one by one to groups according to the sorted order. For each composite cell, we assign it to the group who has the smallest number of records so far. This simple solution may not achieve the optimal results, i.e., the minimum standard deviation on the workloads of all reduce tasks, it is adequate for our problem since it computes roughly balanced groups with negligible computational overhead.

The detailed steps are as follows. Before the `map` function gets executed, the composite cells are grouped based on the their numbers of records collected in Job 2. In the `map` function, histogram h is first transformed to (m_h, b_h) with its approximation error values C_h. Then the dual and centroid keys of the histogram is computed. Next, for each cell $g \in G$ and each projection j, $LB_{normal}(h, g^j)$ is computed using the formula described in Section 2.2 (cell vertexes can be computed based on the domain and z). If this $LB_{normal}(h, g^j)$ is greater than or equal to ϵ, then g, and effectively G, are pruned for the record. Additionally, a record should be refined with records that lie in the same cell. Hence, for each cell there are two types of records: the *native* ones who lie in the cell and the *guest* ones who do not get pruned for the cell using the lower bound. We use the flags in or out to denote whether a record is native for the cell. We further check any other bounds plugged into the pruning procedure, and distribute the record to the corresponding reducers if all the bounds fail to prune that composite cell for the record. On reducers, a self join is performed on the native records, while a nested-loop join is performed between the native records and the guest records. A chain of EMD lower bounds (discussed in Section 6.1) are computed to further filter dissimilar candidates. Additionally, the reference-UB is used to join two histograms without computing EMD between them, as discussed in Section 2.2.

3.4 Processing Top-k Joins

The difference between the range and the top-k joins is that there is no priori threshold value to prune candidates. Instead, such a threshold needs to be computed and refined during the join procedure. To obtain an initial threshold value, we again leverage Job 1. A small portion (typically 100 records) of the datasets is sampled in the *map* phase and the EMD of pairs among this sample is computed in the *cleanup* phase. The k^{th} smallest EMD values are distributed to workers.

When processing top-k joins, the reference-UB can be employed to prune dissimilar composite cells for a given histogram. The intuition is that for a histogram h, if there are already more than k candidates with upper bounds smaller than threshold value ϵ, then any candidates with a lower bound greater than ϵ can be pruned for this histogram. To implement this idea, in Job 2 mappers, for a histogram h and a reference histogram $\hat{h}$, we compute a feasible flow, denoted as $F(h, \hat{h})$, and

Algorithm 1: Pruning by Upper Bound in Top-k Joins

Input: a histogram h, composite cells $\{G\}$, reference $\{h_{ref}\}$
Returns: a set of un-pruned composite cells

```
1  G, U ← ∅, ε_h ← ∞
2  foreach ĥ do
3      foreach G do
4          U ← U ∪ UB_ĥ(h, G, ĥ)
5      sort U
6      count ← 0
7      foreach UB ∈ U do
8          if count ≥ k then
9              ε_{h,ĥ} ← UB
10             break
11         count ← count + O_G
12     if ε_h > εh, ĥ then
13         ε_h ← εh, ĥ
14 foreach G do
15     flag ← true
16     foreach g ∈ G do
17         flag ← LB_normal(h, g) ≤ ε_h
18     flag ← flag ∧ LB_dual(h, G) ≤ ε_h
19     flag ← flag ∧ LB_centroid(h, G) ≤ ε_h
20     if flag then
21         G ← G
22 return G
```

aggregate the *maximum flow* for each composite cell and each $\hat{h}$, denoted as $\bar{F}_{G,\hat{h}}$ on the reducer. The maximum is necessary as we are only interested in an upper bound for all the records in the composite cell. In Job 3, the upper bound of EMD between a histogram h and a composite cell G can be computed by adding $F(h, \hat{h})$ to the maximum flow $\bar{F}_{G,\hat{h}}$. For $h' \in G$

$$\begin{aligned} UB_{\hat{h}}(h, G) &= F(h, \hat{h}) + \bar{F}_{G,\hat{h}} \\ &\geq F(h, \hat{h}) + F(h', \hat{h}) \\ &\geq F(h, h') \geq EMD(h, h'). \end{aligned}$$

Algorithm 1 lists the steps involved in the pruning. Given h and $\hat{h}$, among all the reference-UBs for different G, we choose one value $\epsilon_{\hat{h}}$ such that it is the smallest value that guarantees there are at least k histograms in the set of composite cells, each of which has reference-UB not greater than $\epsilon_{\hat{h}}$. This essentially indicates that for h there are at least k histograms which will have an EMD not greater than $\epsilon_{h,\hat{h}}$. Formally, let O_G denote the number of records in G,

$$\epsilon_{h,\hat{h}} = \arg\min_{\epsilon} \sum_{G} \{O_G | UB_{\hat{h}}(h, G) \leq \epsilon\} \geq k.$$

Among all $\epsilon_{h,\hat{h}}$ for different reference histograms, we select the *minimum* one ϵ_h, i.e., $\epsilon_h = min_{\hat{h}}\{\epsilon_{h,\hat{h}}\}$ (line 12 to line 13). If a composite cell has any types of lower bounds greater than ϵ_h, it can be pruned for h (line 14 to line 21).

To progressively refine the threshold value during the computation, the reducers in Job 3 now maintain a heap of k pairs of histograms with the smallest EMD values. The largest EMD value in this heap is used to substitute ϵ in the *reduce* phase of the algorithm.

It is clear that though top-k joins can be handled by HEADS-JOIN with MR, the algorithm involves high latency due to the lack of communication (synchronization on the value of ϵ) during the pruning. If the dataset has a skewed distribution, this may result to unsatisfactory pruning results.

4 HEADS-JOIN ON BSP

The MR paradigm is widely adopted, yet it may not be the best option for the EMD similarity join problem. The MR based HEADS-JOIN algorithm faces the inherent drawbacks of MR, i.e., the lack of communication during each job. The BSP paradigm is a prominent alternative on processing large scale datasets. It leverages the distributed main memory by default, and uses *supersteps* to organize local computation and synchronous communications. Recent years have seen an wide adoption of this paradigm on Hadoop[4]. In the following, we show how to devise a HEADS-JOIN algorithm on BSP.

Similar to the three jobs in the MR counterpart, the BSP algorithm has two conceptual phases, i.e., the *preparation phase* and the *pruning and refining phase*.

1) *Preparation phase*. The input histogram datasets are partitioned using the normal-LB and the quantile based grid techniques. Complementary information such as the aggregation errors, the number of records in the composite cells, and the lower key ranges are stored for later access.
2) *Pruning and refining phase*. The partitioned data is pruned using multiple lower bound and upper bound techniques. For the top-k joins, the threshold predicate value ϵ is progressively refined and synchronized throughout the consecutive supersteps.

In the following, we describe in details how these two phases are implemented in the BSP algorithm.

4.1 Preparation Phase

The underlying idea is similar to Job 1 and Job 2 described in Sections 3.1 and 3.2, respectively. However, as the synchronous communication is available in BSP in each superstep, there is no need to use two separate BSP jobs to accomplish the task; one BSP job is employed for the phase and it consists of five supersteps. This job is the same for both the range and the top-k joins.

Algorithm 2 demonstrates the detailed procedure. In BSP, the `superstep` procedure is iteratively executed on parallel workers. Between two consecutive iterations, all the workers wait (line 38) until all the messages are delivered to the corresponding receiving workers. The loop only terminates when all the workers have called the procedure *done*. We use boolean values to organize the five supersteps during the procedure (line 2). As there are only five supersteps and all the operations are at most of linear cost to the input size, this BSP phase is rather lightweight when compared with the second phase.

To start, each worker reads a portion of the data and applies the normal-LB transformation (line 5 to line 10). We process all the histograms in the first superstep as there is no need for communications before all the histograms have been processed. Then, each worker sends the transformed data in each space to one worker (line 11 to line 12), whom acts as the master worker and uses the second superstep to aggregate the domain and the quantile values (line 15 to line 16). Since there are multiple spaces, multiple master workers are involved. After all the domains and quantile values are computed, these master workers send these values to all the workers (line 17 to line 18). Next, each worker uses the third superstep to assign the original input histogram to the corresponding composite cell, aggregate locally the number of records, the aggregation errors, the dual key ranges, the centroid key ranges, and the maximum feasible flows, for all the composite cells encountered (line 20 to line 23). After all the input histograms are processed, the aggregated information for the composite cells are sent to one master worker (line 24), whom uses the fourth superstep to compute the global aggregated information for all composite cells (line 27 to line 29). All the aggregated information is written out and the messages on how the composite cells should be grouped together (for load balancing) are sent to all the workers (line 30 to line 31). In the last superstep, all the workers use the grouping information to partition data and write it out to the disk as the input for the second phase (line 34 to line 37).

4.2 Pruning and Refining Phase

In this phase, we conduct the actual pruning and the refining. Each worker reads histograms assigned to a particular group e_G, and read the aggregated information for all the composite cells, to prune composite cells, i.e., other workers, from each histogram. As described in Section 3, histograms in the same composite cell need to be refined first. Then, we conduct histogram-worker pruning. Given a histogram h, for each composite cell group e_G, we use the normal, the dual, and the centroid lower bound to see if the h needs to be refined with the histograms in that group e_G. If so, the histogram is sent to the worker that is responsible for refining e_G. When processing top-k joins, reference-UB can be employed to prune G as described in Section 3.4.

A straightforward approach on organizing the superstep procedure is to conduct the whole pruning in one big superstep and then refine all the received histograms with the input histograms. However, doing so disables updating the joining threshold for a given k as more histograms are processed. A better way is to progressively perform the refinement between the received and the input histograms. That is, we use finer supersteps and conduct the refinement after a certain number of

4. Apache Hama, http://hama.apache.com

Algorithm 2: Preparation BSP Job

```
1  bsp-setup
2  transformed, aggregated, partitioned, grouped ← false
3  superstep
4  if ¬transformed then
5      M, B ← ∅
6      foreach h do
7          foreach j ← 1 to p do
8              CDF_h ← {w_h}, L^{v_j}
9              Φ(μ, σ^2) ← CDF_h
10             M_j ← 1/σ , B_j ← −μ/σ
11     foreach j ← 1 to p do
12         send message M_j and B_j to worker j
13     transformed ← true
14 else if ¬aggregated then
15     M, B ← read message
16     Q_m, Q_b ← M, B
17     foreach worker do
18         send message Q_m, Q_b to worker
19     aggregated ← true
20 else if ¬partitioned then
21     Q_m, Q_b ← read message
22     foreach h do
23         G_h, C_h, λ_h, ω_h ← h, Q
24     send message {(G, O_G, C_G, λ_G, ω_G)} to master
25     partitioned ← true
26 else if ¬grouped ∧ isMaster then
27     G, O_G, C_G, λ_G, ω_G ← read message
28     e_G ← G, O_G, C_{e_G} ← C_G, e_G, λ_{e_G}, ω_{e_G} ← λ_G, ω_G, e_G
29     write (e_G, C_{e_G}, λ_{e_G}, ω_{e_G})
30     foreach worker do
31         send message e_G to worker
32     grouped ← true
33 else
34     e_G ← read message
35     foreach h do
36         write (e_G, h)
37     done
38 synchronize
```

input histograms are sent. This could be beneficial in terms of the memory and the network bandwidth usage because at any given time, the number and the length of messages sent are relatively small. Since the threshold value can be sent to all the workers via messages, the algorithm naturally supports the progressive refinement on the threshold value such that all workers can use the globally smallest threshold value.

The histogram-worker pruning may result in some worker receiving no messages. This is undesired as the synchronization in the superstep suggests all workers cannot continue to the next superstep until the slowest worker finished. Compared to refining the received histograms with the input histograms, which possibly involves EMD computations, the histogram-worker pruning is substantially cheaper. We therefore allow workers to wait on pruning (lightweight) instead of wait on refining (heavy). Hence, each worker maintains a counter for the messages sent and keep processing the input until the counter reaches a predefined number $batch$. After $batch$ number of messages are generated,

Algorithm 3: Pruning and Refining BSP Job

```
1  bsp-setup
2  e_G, C_G, λ_G, ω_G ← HDFS
3  native, guest, final ← false
4  O_{e_G}, guestCount, guestDoneCount ← 0
5  superstep
6  if ¬native then
7      self-join all input h ∈ H_in
8      if topk then
9          ε_local ← k^th smallest EMD to worker
10         foreach worker do
11             send message ε_local to worker
12     native ← true
13 else if ¬guest then
14     M ← ∅
15     foreach h offset by guestCounter do
16         foreach e_G do
17             flag ← false
18             foreach G ∈ e_G do
19                 eachFlag ← true
20                 foreach g ∈ G do
21                     if LB_normal(h, g) > ε then
22                         eachFlag ← false
23                         break
24                 if LB_dual(h, G) > ε ∨ LB_cent(h, G) > ε then
25                     eachFlag ← eachFlag ∧ false
26                 flag ← flag ∨ eachFlag
27             if flag then
28                 M ← (e_G, h)
29         guestCounter++
30         if guestCounter ≥ batch then
31             break
32     if guestCounter = Σ_{G∈e_G} |C_G| then
33         guest ← true
34         foreach worker do
35             send message (me, guestDone) to worker
36     if M ≠ ∅ then
37         foreach (e_G, h) ∈ M do
38             send message (e_G, h) to worker responsible for e_G
39 H_out, guestDoneCount, {ε} ← read message
40 if guestDoneCount = #ofworkers then
41     done
42 if topk then
43     ε ← min({ε}, ε_local)
44 join H_in and H_out, update ε if topk
45 foreach worker do
46     send message ε to worker
47 synchronize
```

all the messages destined to the same worker will be packed into one single long message and sent accordingly. This essentially incorporates the improvement of the CGM [10] paradigm. There is a trade-off when choosing the proper $batch$ value: a large value produces fewer larger messages, yet it delays the refinement on the threshold value; a small value produce more smaller messages which may introduce race and contention over the network, yet it aggressively refines the threshold value. The best value is subject to various factors: the network bandwidth, speed, the dataset distribution, etc; choosing which is beyond the scope of this paper.

Algorithm 3 lists the detailed steps in this phase. Again, we use three boolean values to indicate the three different subtasks in the phase, i.e., refining histograms in the same partition (line 6 to line 12), pruning and sending histograms to other partitions (line 13 to line 38), and refining histograms received from other partitions (line 39 to line 46). In the first subtask, performing self-joins on histograms within the same partition is carried out independently. However, in the second subtask, the same number of sent messages may render the processing rate different on different workers. For example, if a worker X prunes many workers for many histograms, X may end up finishing all pruning in early supersteps. However, X cannot call *done* before the last worker finishes pruning, since it is possible that there are still histograms on the other workers which need to be refined with histograms on X. Hence, a variable $guestDoneCount$ is maintained on every worker for counting the number of workers that have completed pruning for all the histograms in the partition assigned to them. When all the workers have reported the completion of pruning, all the workers call *done* and the loop is terminated. We cannot use $H_{out} = \emptyset$ as the termination condition either, since it is entirely possible that during some intermediate supersteps one worker does not receive any incoming messages but still engage in the refinement later on.

When processing top-k joins, additional messages are sent to report to all workers the current known k^{th} smallest EMD threshold value (line 10 to line 11 and line 45 to line 46). This way, all workers can always use the globally smallest threshold seen so far.

5 HEADS-JOIN ON SPARK

The BSP paradigm overcomes the lack of synchronization problem, yet it has its own inherent disadvantages. For example, it forces synchronization between each superstep, essentially prolonging each superstep by always waiting for the slowest worker. Moreover, it sends a large number of messages over the network, which may result in contention on certain workers. In this section, we further explore the Spark (RDD) paradigm, which has been demonstrated in existing studies to outperform MR in many different tasks.

Spark supports equi-joins (joining records with the same key) between two RDDs via its native interfaces. However, these equi-join oriented interfaces cannot be applied to the similarity joins where the predicate allows arbitrary functions. Therefore, we need to implement HEADS-JOIN using other interfaces defined on RDD itself. Specifically, four major interfaces are involved: `map` which applies a function to each record, `groupBy` which merges records with the same specific value, `collect` which transfers an RDD to a local collection on a single machine, and `reduce` which aggregates all records to obtain a single value. Among these interfaces, `map` and `groupBy` generate new RDDs, while `collect` and `reduce` convert RDDs to local value.

Algorithm 4: The `prep` phase in Spark

Input: $data$: RDD
Returns: $grids$, $error$, $bound$, $assignment$

1 $trn \leftarrow data$.**map** $\{h \rightarrow (M, B)\}$
2 $grids \leftarrow trn$.**map** $\{(M, B) \rightarrow$ compute percentile $\}$.**collect**
3 $error \leftarrow data$.**map** $\{h \rightarrow G_h\}$.**groupBy** $(G_h)\{i$.**map** $\{h \rightarrow C\}$.reduce$\}$.**collect**
4 $bound \leftarrow data$.**map** $\{h \rightarrow G_h\}$.**groupBy** $(G_h)\{i$.**map** $\{h \rightarrow \lambda\}$.reduce$\}$.**collect**
5 $native \leftarrow data$.**map** $\{h \rightarrow G_{h,in}\}$.**reduce**
6 $guest \leftarrow data$.**map** $\{h, grids, error, bound \rightarrow G_{h,out}\}$.**reduce**
7 $assignment \leftarrow native, guest$

Similar to the BSP algorithm, we have two phases for HEADS-JOIN on Spark: one for preparing the grids (denoted as `prep`); and one for filtering and refining (denoted as `refine`). For both range joins and top-k joins, the procedure is largely the same, with top-k joins having extra pruning to perform. In the remainder of this section, we elaborate based on the range joins.

The goal of the `prep` phase is to 1) construct the grid structures using multiple projection vectors, 2) aggregate the error and bound values for each composite cell, and 3) assign composite cells to workers based on the workload estimation. Different from both the MR and the BSP algorithms, here we estimate the workloads on each worker using the *product of the numbers of native and the guest records*, instead of the number of the native records. Remind when refining, the workload contains a self join on the native record and a nested loop on the native and the guest records. As the number of guest records is typically larger than the number of native records, multiplying the number of guest records with the number of native records should give a more accurate estimation. To compute this estimation, we need an extra step in the Spark algorithm to count the number of guest records before shipping those guests records to their destined workers. Thanks to Spark's capability of persisting intermediate RDD, this extra step will generate the pruning results that can be reused in the next phase. This can be elegantly achieved as demonstrated in Algorithm 4. Clearly, lines 1 to 2 accomplish the task corresponding to Job 1 in the MR implementation; lines 3 to 5 accomplish the task corresponding to the Job 2. Lines 6 and 7 are the additional step we take to improve the estimation on the workloads, i.e., counting the guest records for each composite cell and assigning composite cells to groups based on the product of the native and the guest records.

The `refine` phase then processes the output of the *prep* phases. As demonstrated in Algorithm 5, this corresponds to the Job 3 in the MR algorithm. Specifically, line 1 finds the relevant composite cells by pruning using the normal-LB, dual-LB, and the centroid-LB. Line 2 finds the worker responsible for each composite cell and distributes the records to the corresponding workers with the `groupBy` operation. Last, line 3 conducts the EMD join with a chain of pruning lower bounds as described in Section 3.3.

Algorithm 5: The `refine` phase in Spark

Input: $data$: RDD, $grids$, $error$, $bound$, $assignment$
Returns: $pair$: RDD

1 $cells \leftarrow data$.**map** $\{h, grids, error, bound \rightarrow G_{h,in}, G_{h,out}\}$
2 $group \leftarrow cells$.**map** $\{G, assignment \rightarrow e_G\}$**groupBy** (e_G)
3 $pair \leftarrow group$.**reduce** {chained filtering and joining}

6 Related Work

6.1 Lower Bounds of EMD

Devising computationally cheap lower bounds for EMD has received much attention from different fields. The random projection technique [7] maps histograms to one-dimensional space, the dimensionality reduction technique [25] employs a reduction matrix and ground distance adjustment to largely decrease the number of bins, and the independent minimization technique [3] relaxes the optimization constraint in computing the EMD. Transformation based methods transform high-dimensional histograms to lower dimension and perform a computationally cheap operation (rather than EMD) on these transformed values to provide the lower bound of the EMD. The normal-LB, the dual-LB, and the centroid-LB belong to this family.

6.2 Processing Join Using Parallel Paradigms

As described in Section 2.3, three emerging computation paradigms in Hadoop include MapReduce, BSP, and RDD(Spark). We briefly review the related studies on these paradigms.

On MapReduce, the join operations have been the focus of many studies [6], [17], [16], [24], [14]. However, effective pruning techniques such as prefix-filtering [24] and inverted index [16] are not applicable in our problem because they leverage the property of sparse dimension in the data objects that is absent here.

On BSP [23], parallel join has been previously investigated [4]. Our study differs from the existing BSP join solutions in that i) our solution is specifically designed for the EMD similarity join, leveraging the lower bound techniques of EMD; ii) our solution is targeted to the Hadoop platform, where typically shared-nothing clusters are used and a distributed file system is available. Other closely related parallel frameworks such as LogP [8] and CGM [10] further relaxes and improves upon BSP. Specifically, LogP replaces the synchronization barrier in BSP with a more constrained point-to-point message passing mechanism, aiming at limiting the network communication cost; CGM replaces the synchronization with a global communication round where all the information sent between two workers are packed into one long message. As such improvement focuses on the communication and does not affect the computation, its enhancement is orthogonal to designing the algorithms.

On Spark, thanks to its more general and expressive interfaces than MR, those techniques designed for MR may be applied (by simulating MR with `map`, `groupBy`, and `reduce` in Spark). However, the design concern may shift as Spark relies on the main memory on distributed machines instead of HDFS, and it is more flexible than MR when synchronization is needed. We have leveraged this difference in our Spark algorithm of HEADS-JOIN with the more effective estimation on the workloads of workers (as described in Section 5. Recent studies on Spark focuses on streaming processing [29] and graph processing [11], which extends the interfaces of Spark to support more domain specific tasks.

6.3 State-of-the-Art Metric Space Similarity Join

Quickjoin [13] divides the space using *pivot* points such that data objects are assigned to nearest pivot to form multiple clusters. The data objects in the margin of clusters may have small distances hence should also be joined to guarantee the completeness. MRSimJoin [22] is the MR implementation. In MRSimJoin, mappers sample the pivots and assign data objects to clusters and then reducers join the records in the same cluster and the same margin. We implemented a Quickjoin algorithm using BSP. Briefly, the pivots are sampled in a superstep and sent to all workers, using which they prune histograms from regions, and each worker is responsible for refining histograms distributed to a region.

The major problem of Quickjoin is the excessive number of distance computations since each data object needs to find its nearest pivot. The lower bounds of EMD cannot be integrated into this procedure since i) most lower bounds are not metric; ii) even a metric lower bound may not preserve the *locality of EMD*, i.e., for an object, its nearest pivot in terms of the lower bound is not necessarily its nearest pivot in terms of EMD. Furthermore, the parallel implementation of Quickjoin can be vulnerable to skewed data distribution. For example, in MRSimJoin if a reducer receives a large number of objects, it further samples pivots and partitions those objects. This is inefficient in our problem as it introduces more EMD computations.

7 Performance Evaluation

We present our experimental study in this section. **Please note the figures in this section are presented in colors.**

7.1 Environment and Methods

We conduct all the experiments on a research cloud platform with OpenStack. Up to 16 virtual machine instances are employed, each of which equips with four 2.8GHz virtual cores and 16GB memory. The network bandwidth between virtual machines is up to 600Mbps. The JDK version is 1.8.0. We use Hadoop 2.6.0 (YARN and HDFS) and leverage YARN to manage the resource allocation for all the three paradigms.

We have three algorithms of HEADS-JOIN on different computation paradigms, i.e., MR, BSP, and Spark,

denoted as **mh**, **bh**, and **sh**, respectively. The MR algorithm is implemented over Apache Hadoop MR 2.6.0 with Java 1.8, the BSP algorithm is implemented over Apache Hama 0.7.0-SNAPSHOT with Java 1.8, and the Spark algorithm is implemented over Apache Spark 1.3.0 with scala 2.11.3. We fixed several bugs in Hama when interacting with YARN so that all three algorithms can be managed and scheduled by YARN. We also implement Quickjoin over the three paradigms, denoted as **mq**, **bq**, and **sq**, respectively. Additionally, we implement a naive block nested loop join (BNL) over Spark, denoted as **sb**, which should serve as the upper bound of performance for both HEADS-JOIN and Quickjoin.

To compare the seven methods, we focus on the elapsed time which includes both the CPU time and the cost of file system IO as well as network IO. Since the EMD similarity joins are rather computation-intensive, the CPU time constantly dominates the overall cost in all our experiments. Moreover, the elapsed time is also the main indicator for the real economical cost on a commercial cloud platform such as Amazon Web Services or Microsoft Azure. For two BSP methods, we additionally report the number of messages exchanged.

7.2 Datasets and Configurations

We report the performance of all the methods on three datasets, among which the MIRFlickr dataset [15] and Million Songs [5] dataset are real and the other one is synthetic. For the MIRFlickr dataset, we use various CBIR features, such as the MPEG-7 color descriptors and edge descriptors. Since the relative performance gain is quite consistent over different features, we show the results on MIRFlickr with Color Layout Descriptor and denote the feature dataset as A. For the Million Songs dataset, we use the timbre features and denote the feature dataset as B. For the synthetic dataset, we randomly (in a uniform fashion) generate 24-dimensional vectors as histograms and a 48-dimensional vector as the 2-dimensional bin locations for the 24 bins. We denote this synthetic dataset as dataset C. Table 1 shows the cardinality and the number of bins of all the datasets and Table 2 shows the EMD distribution among the datasets, where the last column $D_{n,u}$ is the Kolmogorov-Smirnov statistic (ranging from 0 to 1) between the distances of a uniform distribution and a sample of EMD in the corresponding dataset. The larger this value suggests a more skewed distribution of the dataset.The presented datasets may appear small when compared to the typical datasets involved in a Hadoop based study. However, as demonstrated in Fig. 7, the EMD similarity joins are so computation-intensive that a single machine takes hours to join even the smallest datasets. With the limit of our computation resource (16 virtual machines), to measure the performance within a reasonable amount of time (6 hours per run), the size of datasets needs to be restricted.

The parameters of the frameworks are configured as follows. For HEADS-JOIN , we set the p and z to 4 and 4 for all the datasets. The optimal values of these parameters are subject to various factors, such as the data distribution, the dimensionality, the number of machines, and the hardware specification of the machines. Tuning the parameters is beyond the scope of this paper. For Quickjoin, we adjust the number of pivots to match the number of the available reducers or workers. For all BSP algorithms, $batch$ is set to 100. The join predicate ϵ in the range joins is varied according to Table 2 to produce a reasonable selectivity. The predicate k in the top-k joins varies from 1 to 10000, where 10 is the default value.

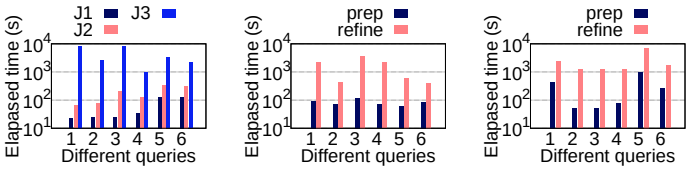


Fig. 1. Time breakdown in different implementations

TABLE 1
Datasets in the emperical study

Dataset	Cardinality	n	ϵ
MIRFlickr (A)	**62.5k**, 125k, 250k, 500k, 1m	16	0.002
Million Songs (B)	**32K**, 64K, 128K, 256K, 512k	32	0.2
Synthetic (C)	12.8k(1X), **25.6k(2X)**, 51.2k(4X), 102.4k(8X), 204.8k(16X), 409.6k(32K), 819.2k(64X)	24	1.0

7.3 Performance Break-down of Different Phases

We present the detailed time cost of each of different phases in the HEADS-JOIN algorithms in this section. The results of executing different queries are illustrated in Fig. 1. Here, queries 1 to 6 corresponds to the top-k joins and range joins on datasets A, B, and C, respectively. In the figure, `J1`, `J2`, and `J3` corresponds to the three MR jobs in the MR algorithm. As expected, `J3` always dominates the overall cost as `J1` and `J2` are at least an order of magnitude cheaper than `J3` and therefore negligible in the overall cost. Similarly, the preparation phase `prep` in the BSP algorithm is two orders of magnitude cheaper than the actual `bsp` join phase. The preparation phase in Spark, `prep`, is slightly more expensive, constantly sticking at an order of magnitude than the actual `refine` phase. This is because in the Spark algorithm, we estimate the workload of each worker by the product of the native and guest records, which involves an additional step when comparing to the estimation by the number of native records. Overall, the joining phase in all algorithms dominate the time cost of HEADS-JOIN, confirming the efficiency of the involved pruning techniques.

7.4 Performance on Handling Skewed Datasets

The standard deviation of the elapsed time corresponding to the final refinement MR job of HEADS-JOIN and

TABLE 2
EMD distribution of the datasets

Dataset	0.001%	0.01%	0.1%	1%	100%	$D_{n,u}$
A	0.0031	0.0036	0.0049	0.0065	0.0121	0.5004
B	0.3217	0.5429	0.6998	0.9516	2.0650	0.9613
C	1.0491	2.3485	3.4123	4.4482	10.7467	0.3425

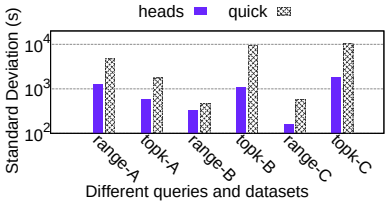


Fig. 2. The standard deviation of completion time among the distributed machines in MR, the smaller the more balanced

Quickjoin on the distributed machines are presented in Fig. 2. Here, all datasets are of the default sizes and parameters are set to their default values. Clearly, the smaller the standard deviation, the more balanced the workloads are on the distributed machines. From Fig. 2, HEADS-JOIN always delivers an order of magnitude more balanced workloads than Quickjoin, suggesting its better performance when handling skewed datasets. The results for BSP and Spark are omitted as in these paradigms the progress on the distributed machines are periodically synchronized; the measure on the workload balancing is less straightforward than in MR.

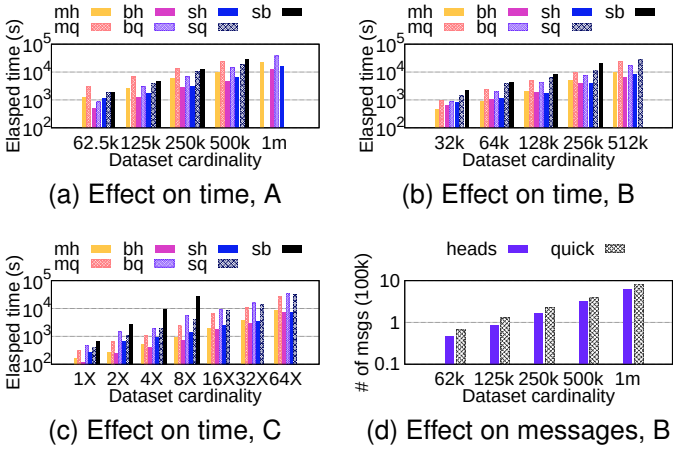
7.5 Effect of Varying Dataset Cardinality

We vary the cardinality of the datasets and measure the performance of all the seven methods.

Comparing the methods in range joins. The results of range joins are presented in Fig. 3. As the size of datasets grows, the cost of all methods climbs linearly with the $|H|^2$ (since it is a join operation). We compare HEADS-JOIN and Quickjoin in three computation paradigms:

- MR. Compared to Quickjoin (mq), HEADS-JOIN (mh) runs from 2.1 to 2.5 times faster on A, from 1.9 to 2.9 times faster on B, and from 2.0 to 3.3 times faster on C.
- BSP. In terms of elapsed time, HEADS-JOIN (bh) is several times faster than Quickjoin (bq): from 1.5 to 3.1 times on A, from 1.4 to 2.7 times on B, and from 5.6 to 7.9 times on C. Moreover, Quickjoin sends more messages than HEADS-JOIN , e.g., 1.3 to 1.5 times on B.
- Spark. Similarly, HEADS-JOIN (sh) is always faster than Quickjoin (sq): from 1.7 to 3.5 times on A, from 1.6 to 3.6 times on B, and from 1.8 to 4.3 times on C. Both methods outperforms the BNL method (sb), while HEADS-JOIN usually beats BNL by an order of magnitude.

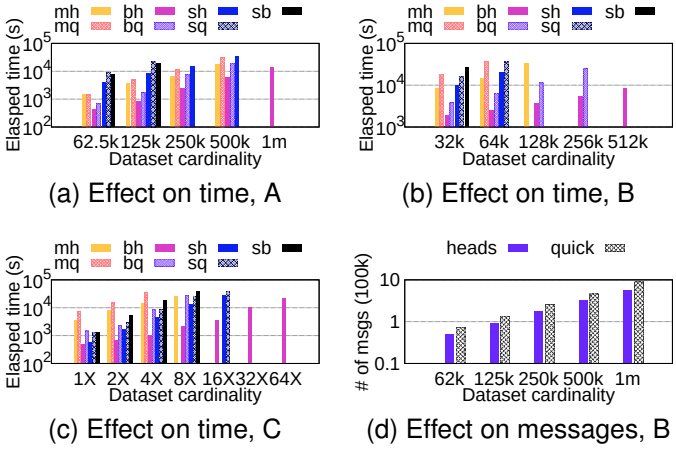
Comparing the methods in top-k joins. The results of top-k joins are presented in Fig. 4. As expected, all



(a) Effect on time, A (b) Effect on time, B

(c) Effect on time, C (d) Effect on messages, B

Fig. 3. Range joins with varying dataset cardinalities



(a) Effect on time, A (b) Effect on time, B

(c) Effect on time, C (d) Effect on messages, B

Fig. 4. Top-k joins with varying dataset cardinalities

methods take more time when the dataset grows. We compare HEADS-JOIN and Quickjoin as follows:

- MR. While on A, Quickjoin achieves a competitive performance when dataset is small (62.5k), it is beaten by HEADS-JOIN on all other datasets. Specifically, the speedup from Quickjoin of HEADS-JOIN is from 1.1 to 1.7 times on A, from 1.1 to 2.6 times on B, and from 1.5 to 2.4 times on C. Such an advantage is increasing when dataset grows, as Quickjoin fails to return results on medium size datasets within 6 hours.
- BSP. In terms of elapsed time, the speedup factors of HEADS-JOIN from Quickjoin are: from 2.1 to 3.2 times on A, from 2.3 to 4.7 times on B, and from 6.2 to 13.0 times on C. Remarkably, the BSP algorithm of HEADS-JOIN is the only method that manages to complete joins on large datasets. In terms of number of messages, Quickjoin constantly sends 1.5 times more messages than HEADS-JOIN .
- Spark. HEADS-JOIN is always several times faster than Quickjoin: from 2.0 to 2.4 times on A, around 1.8 times on B, and from 2.1 to 2.3 times on C.

Comparing the computation paradigms. We compare the three computation paradigms in Fig. 3 and Fig. 4.

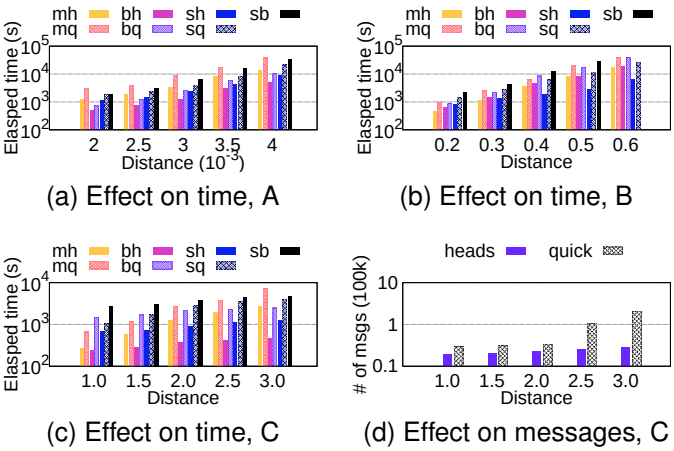
Note this is a rough comparison because: 1) methods are implemented in different programming languages; 2) the implementation of the paradigms are based on different packages, e.g., protocol buffer versus kryo serialization, etc. When processing range joins, there is no clear winner in a general sense and the difference between paradigms are rather steady when the dataset grows. This is because the join cost is dominated by the CPU computation, there is virtually little difference since all methods follow HEADS-JOIN to distribute data and conduct the refining procedure. Both the disk based method, i.e., MR, and the memory based method, i.e., BSP and Spark, need to load the data into main memory and conduct the expensive computation. However, when processing top-k joins, BSP is clearly the best method as it is the only one that is able to perform top-k joins on large datasets. This is because BSP supports the frequent synchronizations between workers when performing the joining procedure, which quickly lowers the joining threshold for a given k. The winner between the MR and the Spark algorithms change from datasets to datasets: Spark is beaten by MR on A and B while beats MR on C. The reason for this lies in the difference between the MR and the Spark methods and is twofold.

1) The Spark method estimates the workloads by additionally considering the number of guest records; this better balances the workloads by paying some extra cost. For larger datasets such as those from A and B, the extra cost outweighs the benefits of more balanced workloads.
2) Spark splits the main memory into two parts: one part for persisting intermediate RDDs to avoid repeated computations, and the other part for the general heap usage. Despite the fact that the datasets involved are relatively small, the intermediate RDDs and temporary objects can be rather large. When some RDDs cannot be persisted in the main memory, they will be re-computed when referenced in the future. Such repeat computations may result in redundant cost in the measurement.

Comparing two join operations on the same dataset. From Fig. 3 and Fig. 4, it is clear that the top-10 joins take more time than the range joins with default ϵ values for all the methods on all datasets. The reason is that the estimated k^{th} smallest distance is likely to be much larger than the default ϵ value, resulting in worse pruning effects. This is especially true for methods that are not able to synchronize regularly, e.g., the MR and Spark methods. For the BSP algorithm of Quickjoin, the progressively refined threshold value in top-k join also introduces problems when assigning records to partitions: a record is likely to be assigned to multiple partitions as it lies right between many partitions.

7.6 Effect of Varying Query Parameter

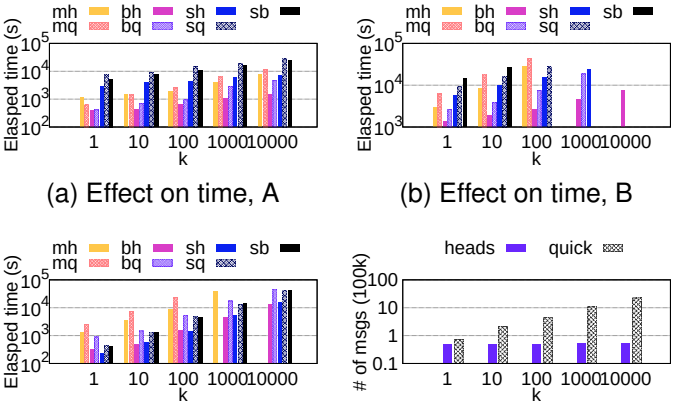
We vary the parameters of the join predicate, i.e, ϵ in the range joins and k in the top-k joins, and measure the performance of all the seven methods.



(a) Effect on time, A (b) Effect on time, B

(c) Effect on time, C (d) Effect on messages, C

Fig. 5. Range joins with varying range distances



(a) Effect on time, A (b) Effect on time, B

(c) Effect on time, C (d) Effect on messages, B

Fig. 6. Top-k joins with varying k

Comparing the methods in range joins. The results of range joins with varying ϵ are shown in Fig. 5.

- MR. The speedup factor of HEADS-JOIN from Quickjoin is around 3 times on all the datasets, and it is relatively steady to the increasing values of ϵ.
- BSP. The speedup factors of HEADS-JOIN from Quickjoin are: from 1.9 to 2.1 times on A, from 1.7 to 2.0 times on B, and from 4.6 to 6.1 times on C. Moreover, Quickjoin sends 1.6 to 7.3 times more messages than HEADS-JOIN on C. While HEADS-JOIN sends marginally more messages when ϵ grows, Quickjoin sends exponentially more messages.
- Spark. The speedup factors of HEADS-JOIN from Quickjoin are: from 1.8 to 2.3 times on A, from 2.4 to 4.0 times on B, and from 2.4 to 3.1 times on C.

Comparing the methods in top-k joins. The results of range joins with varying ϵ are shown in Fig. 6.

- MR. Quickjoin manages to beat HEADS-JOIN on A when only one pair is needed ($k = 1$), but is from 1.1 to 1.7 times slower than HEADS-JOIN when k grows larger. On both B and C, HEADS-JOIN is always faster than Quickjoin: from 1.9 to 2.8 times on B, and from 2.3 to 2.6 times on C.

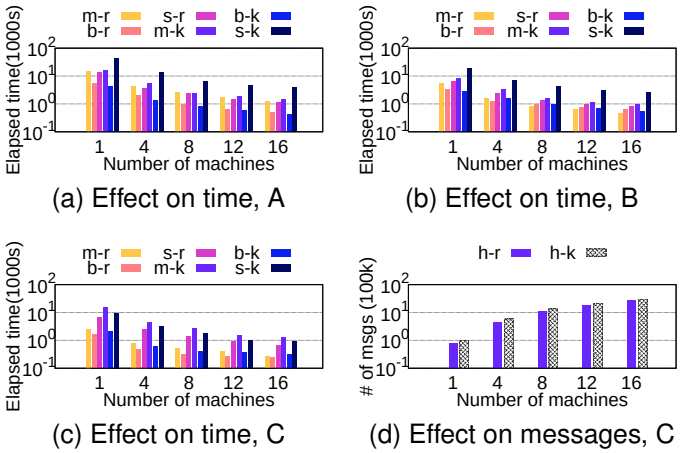


Fig. 7. Scaling out HEADS-JOIN

- BSP. In terms of the elapsed time, the speedup factors of HEADS-JOIN over Quickjoin are: from 2.2 to 3.2 times on A, from 3.1 to 4.1 times on B, and from 2.8 to 3.4 times on C. In terms of the number of messages, Quickjoin sends 1.4 to 41.3 times more messages than HEADS-JOIN . When k grows, for HEADS-JOIN the number of messages grows by a small constant factor, while for Quickjoin it grows linear to k.
- Spark. The speedup factors of HEADS-JOIN over QuickJOin are: from 2.5 to 3.9 times on A, around 2.2 times on B, and from 1.8 to 2.5 times on C.

Comparing the computation paradigms. Again, for range joins, there is no clear winner and all methods take longer time when ϵ grows. While the MR algorithm is the most sensitive method towards the growth of k, the Spark algorithm is the least sensitive one. This is because Spark algorithms involve the extra cost on estimating the workloads using the product, which stays the same for different ϵ values. With such a constant overhead, the overall cost of Spark appears to be less sensitive. For top-k joins, the BSP algorithm is clearly the winner, thanks to its frequent synchronization on the joining threshold.

7.7 Scaling Out

We vary the number of virtual machines in the cluster and measure the performance of all HEADS-JOIN methods. All the datasets and ϵ values are the default ones (the bold ones in Table 1), and k is set to 10. The results are demonstrated in Fig. 7 and Table 3, where `m-r`, `b-r`, `s-r`, `m-k`, `b-k`, and `s-k` represent the MR range joins, the BSP range joins, the Spark range joins, the MR top-k joins, the BSP top-k joins, and the Spark top-k joins. When there are more machines, the time needed for the join operation is reduced. Specifically, when the number of machines increases from 1 to 16, the join operations speeds up by up to **12.8** times (note the logscale y axes in Fig. 7). Moreover, the speedup of all the methods on every dataset follows a quasi-linear manner with different nonlinear overheads. It is not a fully linear relation because more workers inevitably introduces more records distributed over the network (e.g., the messages sent in BSP methods as shown in Fig. 7d), resulting in higher overhead costs.

TABLE 3
Detailed scale-out speedup ratio corresponding to Fig. 7

Dataset	Paradigm	Join	Speedup on Number of Nodes				
			1	4	8	12	16
A	MR	range	1.00	3.38	5.38	8.17	11.20
		top-k	1.00	3.10	6.95	8.89	11.43
	BSP	range	1.00	2.67	5.70	8.21	10.83
		top-k	1.00	3.03	4.84	6.71	9.61
	Spark	range	1.00	3.60	5.68	9.47	12.15
		top-k	1.00	3.31	6.73	9.56	11.50
B	MR	range	1.00	3.45	6.29	8.23	11.21
		top-k	1.00	2.61	5.17	7.37	8.52
	BSP	range	1.00	2.60	4.46	6.40	8.95
		top-k	1.00	2.13	4.00	6.06	8.78
	Spark	range	1.00	2.62	4.60	6.53	10.13
		top-k	1.00	2.75	4.42	6.45	10.10
C	MR	range	1.00	3.24	4.93	6.45	9.70
		top-k	1.00	3.50	5.55	9.86	11.78
	BSP	range	1.00	3.34	4.95	6.31	12.57
		top-k	1.00	3.54	5.11	8.49	12.79
	Spark	range	1.00	2.66	4.90	7.19	9.82
		top-k	1.00	2.95	5.08	9.00	10.49
		Average	1.00	3.00	5.15	7.15	10.70

7.8 Summary

While both Quickjoin and HEADS-JOIN outperform the naive NBL join, HEADS-JOIN constantly reduces the running time of Quickjoin by up to an order of magnitude. For the BSP algorithms, HEADS-JOIN sends up to 41 times fewer messages than Quickjoin thanks to the CGM's mechanism on batching a large number of short messages into several long messages. Moreover, HEADS-JOIN enjoys the quasi-linearly horizontal scale-out property.

8 CONCLUSION AND FUTURE WORK

We proposed HEADS-JOIN, a novel framework for processing the EMD similarity joins based on the Hadoop platform. HEADS-JOIN employs multiple computationally cheap lower bounds to prune and partition data. We tackled both the range and the top-k joins by designing efficient algorithms using the MR, the BSP, and the RDD(Spark) paradigms. We conducted experiments on various real datasets to evaluate efficiency of HEADS-JOIN. As demonstrated by the results, HEADS-JOIN outperforms the state-of-the-art metric similarity join technique, i.e., Quickjoin, by up to an order of magnitude and it scales out well. In the future, we would like to investigate employing GPUs to further parallelize the EMD computation itself to scale HEADS-JOIN to even larger scale datasets.

ACKNOWLEDGMENTS

This work is supported in part by the Australian Research Council (ARC) Discovery Project DP130104587, National High-Tech R&D (863) Program of China (2013AA01A213), Natural

Science Foundation of China (61433008, 61373145, 61170210, U1435216), Chinese Special Project of Science and Technology (2013zx01039-002-002). Dr. Zhang and Dr. Buyya are supported by the ARC Future Fellowships Projects FT120100832 and FT120100545 respectively. Prof. Chen is supported by the Fundamental Research Funds for the Central Universities (Grant No. 2015ZZ029).

REFERENCES

[1] M. E. Ali, E. Tanin, R. Zhang, and L. Kulik, "A motion-aware approach to continuous retrieval of 3d objects," in *ICDE*, 2008.
[2] D. Applegate, T. Dasu, S. Krishnan, and S. Urbanek, "Unsupervised clustering of multidimensional distributions using earth mover distance," in *KDD*, 2011.
[3] I. Assent, A. Wenning, and T. Seidl, "Approximation techniques for indexing the earth mover's distance in multimedia databases," in *ICDE*, 2006.
[4] M. Bamha and M. Exbrayat, "Pipelining a skew-insentitive parallel join algorithm," *Parallel Processing Letters*, 2003.
[5] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *ISMIR*, 2011.
[6] S. Blanas, J. M. Patel, V. Ercegovac, and J. Rao, "A comparison of join algorithms for log processing in mapreduce," in *SIGMOD*, 2010.
[7] S. Cohen and L. Guibas, "The earth mover's distance: Lower bounds and invariance under translation," Stanford University, DTIC Report, 1997.
[8] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser, "Fast parallel sorting under logp: from theory to practice," *Protability and Performance for Parallel Processing*, 1993.
[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.
[10] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scala parallel geometric algorithms for coarse grained multicomputers," in *SCG*, 1993.
[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graphx processing in a distributed dataflow framework," in *OSDI*, 2014, pp. 599–613.
[12] J. Huang, R. Zhang, J. Chen, and R. Buyya, "Melody-join: Efficient earth mover's distance similarity join using mapreduce," in *ICDE*, 2014.
[13] E. H. Jacox and H. Samet, "Metric space similarity join," *ACM TODS*, 2007.
[14] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *PVLDB*, 2012.
[15] B. T. Mark J. Huiskes and M. S. Lew, "New trends and ideas in visual concept detection: The mir flickr retrieval evaluation initiative," in *MIR*, 2010.
[16] 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.
[17] A. Okcan and M. Riedewald, "Processing theta-join using mapreduce," in *SIGMOD*, 2011.
[18] Z. Ren, J. Yuan, and Z. Zhang, "Robust hand gesture recognition based on finger-earth mover's distance with commodity depth camera," in *MM*, 2011.
[19] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International Journal of Computer Vision*, vol. 40, pp. 99–121, 2000.
[20] B. E. Ruttenberg and A. K. Singh, "Indexing the earth mover's distance using normal distributions," *PVLDB*, 2012.
[21] M. A. Ruzon and C. Tomasi, "Edge, junction, and corner detection using color distributions," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 2001.
[22] Y. N. Silva and J. M. Reed, "Exploiting mapreduce-based similarity joins," in *SIGMOD*, 2012.
[23] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 1990.
[24] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD*, 2010.
[25] M. Wichterich, I. Assent, P. Kranen, and T. Seidl, "Efficient emd-based similarity search in multimedia database via flexible dimensionality reduction," in *SIGMOD*, 2008.
[26] D. Xu, T.-J. Cham, S. Yan, and S.-F. Chang, "Near duplicate image identication with spatially aligned pyramid matching," in *CVPR*, 2008.
[27] J. Xu, Z. Zhang, A. K. H. Tung, and G. Yu, "Efficient and effective similarity search over probabilistic data based on earth mover's distance," *The VLDB Journal*, 2012.
[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory clutser computing," in *NSDI*, 2012, pp. 2–2.
[29] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *SIGOPS*, 2013.
[30] R. Zhang and M. Stradling, "The hv-tree: a memory hierarchy aware version index," *PVLDB*, vol. 3, no. 1-2, 2010.

Jin Huang Jin Huang is currently a Ph.D. student in the Department of Computing Information Systems at the University of Melbourne, Australia. He received his B.S. in Software Engineering from South China University of Technology in 2011. His research interest includes large-scale similarity analytics, scalable knowledge mining over emerging social data, and efficient indexing techniques.

Rui Zhang Rui Zhang is currently an Associate Professor and Reader in the Department of Computing and Information Systems at The University of Melbourne, Australia. He received his B.S. from Tsinghua University in 2001 and his Ph.D. from National University of Singapore in 2006. His research interest is in areas of high-performance computing, spatial and temporal data analytics, moving object management, indexing techniques and data streams.

Rajkumar Buyya Rajkumar Buyya is a Professor in the Department of Computing Information Systems, a Future Fellow of the Australian Research Council, and the Director of the CLOUDS Laboratory at the University of Melbourne, Australia. He has authored over 450 publications and four text books, and is one of the highly cited authors in computer science worldwide.

Jian Chen Jian Chen is currently a Professor in the School of Software Engineering at South China University of Technology, China. She received her B.S. and Ph.D. degrees, both in Computer Science, from Sun Yat-Sen University, China, in 2000 and 2005 respectively. Her research interests can be summarized as developing effective and efficient data analysis techniques for complex data and the related applications.

Yongwei Wu Yongwei Wu received the PhD degree from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, cloud computing and big data. Dr. Wu has published over 80 research publications and has received three Best Paper Awards.