Fuzzy Joins in MapReduce: An Experimental Study

Ben Kimmett, Venkatesh Srinivasan, Alex Thomo
University of Victoria, Canada
{blk,srinivas,thomo}@uvic.ca

ABSTRACT

We report experimental results for the MapReduce algorithms proposed by Afrati, Das Sarma, Menestrina, Parameswaran and Ullman in ICDE'12 to compute fuzzy joins of binary strings using Hamming Distance. Their algorithms come with complete theoretical analysis, however, no experimental evaluation is provided. They argue that there is a tradeoff between communication cost and processing cost, and that there is a skyline of the proposed algorithms; i.e. none dominates another. We observe via experiments that, from a practical point of view, some algorithms are almost always preferable to others. We provide detailed experimental results and insights that show the different facets of each algorithm.

1. OBJECTIVES

In [1] there are several algorithms proposed for performing "fuzzy join" (an operation that finds pairs of similar items) in MapReduce. The main part of [1] concentrates on binary strings and Hamming distance; this offers the clearest view of the various algorithmic approaches. The algorithms proposed are: *Naive*, which compares every string in the set with every other; *Ball-Hashing*, a family of two algorithms that send strings to a 'ball' of all 'nearby strings' within a certain similarity; *Anchor Points*, a randomized algorithm that selects a set of strings and compares any pair of strings that have a close enough distance to a member of the set; and *Splitting*, an algorithm that splits the strings into pieces and compares only strings with matching pieces.[1]

It is argued in [1] that there is a tradeoff between communication cost and processing cost, and that there is a skyline of the proposed algorithms; i.e. none dominates another. One of our objectives is to see whether we can observe this skyline in practical terms. We observe via experiments that, from a practical point of view, some algorithms are almost always preferable to others. For example in our experiments, Splitting is a clear winner, whereas Ball-Hashing suffers for all distance thresholds except the very small ones. We provide detailed experiments and insights that show different facets of each algorithm.

Another objective we set is to provide implementation optimizations whenever possible. Specifically, we provide optimizations for Naive and Ball Hashing, and clarify details for the others.

[1] *Hamming Code* is another algorithm that [1] considers, however, it is a special case of Anchor Points.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

2. ALGORITHMS AND IMPLEMENTATION

Naive algorithm. This algorithm sends a section of the input to every physical reducer. Each reducer checks every possible pair of strings (out of the set it received) to see if they are within distance d of one another.

More specifically, let $K = J(J+1)/2$ be the number of reducers. They are keyed by (i,j), where $0 \leq i \leq j < J$, thus forming a triangular matrix, where only reducer (i,j) or (j,i) exists. Strings $s \in S$ are hashed to values in $[0,J)$. If a string s hashes to i, it is sent to reducer (i,j) or (j,i), whichever exists, for each $j \in [0,J)$. So, each string is sent to exactly J reducers. Then, [1] suggests that each reducer should exhaustively compare each possible pair of strings from the portion of S it received. This, however, does not need to be so.

Our optimization. Consider strings s and t that both hash to i. They will be sent to reducers $(i,i),(i,i+1),\ldots,(i,J-1)$, and each of these reducers will compare them for similarity. It is clear that only one of the reducers needs to compare s with t, say reducer (i,i); the other reducers should not compare s with t as this would be redundant. More formally, with our optimization, a reducer (i,j) only compares strings that hash to i with those that hash to j. This optimization reduces the amount of work in the reducers by about 2/3. Furthermore this eliminates duplicate output, one of the goals in [1]. The reduction of work by 2/3 is explained as follows. Let n_i and n_j be the numbers of strings hashing to i and j, respectively. An unoptimized reducer does $(n_i^2 + n_j^2 + n_i n_j)/2$ string comparisons and an optimized one does just $n_i n_j/2$. Then the reduction of work is by $(n_i^2 + n_j^2)/2$ comparisons, which is $(n_i^2 + n_j^2)/(n_i^2 + n_j^2 + n_i n_j) = 2/3$ of the work, if we assume $n_i \approx n_j$ in the average case.

Ball-Hashing (BH). This is a family of two algorithms. For these algorithms there is one reducer for each of the possible strings in the universe, so in practice, the reducers are logical rather than physical. A reducer serving string s will receive input strings that are within a ball of a certain

radius r (in terms of number of changed bits) from s. In practice, a physical reducer will likely serve not one string but many.

Ball-Hashing 1 (BH1) sends each input string s to all the reducers serving strings at distance not more than d from s. A mapper reading s, will emit $(s, -1)$ and $(s_1, s), \ldots, (s_k, s)$, where $s_1, \ldots, s_k$ are all the strings obtained from s by changing $i \in [1, r]$ bits. If a reducer receives $(s, -1)$, it infers that s is in the input set. It becomes active and outputs all the (v, s) pairs of similar strings that it received.
Our optimization. The original version of BH1 performs a check for duplicate output in the reducer; a pair (v, s) will be output only if $v < s$ lexicographically. We move this check to the mapper instead, preventing key-value pairs that will only form duplicate output from ever being emitted by a mapper. Specifically, a mapper only emits a key-value pair (s_m, s) if $s_m < s$. This pair will then be received and output by the s_m-keyed reducer. If (s_m, s) is valid, but $s_m > s$, then it is (s, s_m) that will be generated by a mapper. It will be received and output by the s-keyed reducer. The proposed optimization reduces the processing time and communication cost by a constant of roughly two.

Ball-Hashing 2 (BH2) sends each input string s to all the reducers serving strings at a distance of not more than $\lceil d/2 \rceil$ from s. Unlike BH1, every reducer is active and will check for similarity between all the possible combinations of two strings it receives. Pairs that pass the similarity check are then tested to see if they are equidistant from the reducer's "home string".

Splitting (S). The mappers in this algorithm break each string s into $d+1$ equal-length substrings, $s_1, \ldots, s_{d+1}$, and emit $(s_1, s), \ldots, (s_{d+1}, s)$. The rationale for this is that if two strings, $s = s_1 \ldots s_{d+1}$ and $t = t_1 \ldots t_{d+1}$, are at no more than distance d of each other, then they have some substring that is the same for both of them, i.e. there exists $i \in [1, d+1]$, such that $s_i = t_i$. Therefore, there is a reducer that will receive both s and t. Reducers then test each string they receive to see if they are within distance d of all other received strings, similar to the Naive algorithm.

Anchor Points (AP). This is the only randomized algorithm considered, and it has a predicted failure rate of 1/1000 of the time. AP works by choosing a random set of strings ('anchor points') in the universe. If the set is sufficiently large, at least one string in the set can be expected to be within distance $\lceil d/2 \rceil$ of any two strings in the input. The algorithm behaves much like Ball-Hashing in that it sends strings in the input to pools of strings with 'keys' that are slight variants of the string; however, instead of creating every key within distance d or $\lceil d/2 \rceil$ of a string in the input, AP only creates keys that match the strings in the anchor point set and are within distance $2d$ of the input string. All reducers are active, outputting any pair of strings with matching keys that are within distance d of one another.

3. SETUP

In our experiments, the items to be joined were strings of bits, represented as 32-bit integers. For our tests, a universe of strings of length less than 32 bits was used. We ran the algorithms on datasets consisting of:

- The entire universe of 20-bit strings, as a baseline.
- Subsets of the universe of 24-bit strings equal in size to the 20-bit universe (1/16 of the 24-bit universe), as well as 2x (1/8 of ditto) and 4x (1/4 of ditto). The strings in each subset were chosen randomly.
- Subsets of the universe of 28-bit strings equal, double, and quadruple the size of the 20-bit universe respectively (1/256, 1/128, and 1/64).

The results for the 28-bit universe and 24-bit (1/4) are given in the long version of this paper [2]. They do not influence our conclusions in any substantial way.

Each dataset was used on a range of distances from 1, to 1/4 of the length of the strings.

Hadoop Cluster Configuration. The data was processed using Hadoop 1.2.1. In the cluster we used, there is a maximum of 4 map jobs and 4 reduce jobs per machine at any one time. The cluster has 31 machines; each machine has 4 cores (Intel(R) Xeon(R) CPU E5430 operating at 2.66GHz) and 6GB of memory; we have a total of 124 physical cores (one MapReduce node per core). Each Hadoop child process is configured to get 1GB of memory. There are two disks per node. Each disk is a 73GB Hot-Swap 3.5" 10K RPM Ultra320 SCSI HDD capable of a transfer rate of 104 MBytes/sec (measured with dd if=/dev/zero of=test count=1000). The machines are organized in three chassis of 11, 11, and 9 each. Inter-chassis and intra-chassis networking are provided by switches capable of 1Gbit/sec.

4. RESULTS

The gist of our results is summarized below. Then we describe our experiments in detail.

1. We confirm that the shape of Mapper Time (M) accurately represents Communication Cost (C) for Naive, BH1, BH2, and Splitting. We furthermore show that this is essentially true even for AP although not directly suggested by the theoretical analysis.
2. We find that for most algorithms, the main component of the Total Processing Time (T) is the Reducer Time (R). The Mapper Time (M) is much smaller by comparison. The exception is the Ball-Hashing family.
3. BH1 and BH2 are prohibitive for all but small distances because of their enormous communication cost and mapper and shuffle times.
4. AP is fast at low to medium distances. Eventually it becomes a contender with Naive.
5. Splitting is the best algorithm to use across a range of distances.

In a nutshell, our main insight in this paper is that one would always be safe to use Splitting and avoid the Ball-Hashing family. The latter should only be considered for very low distances.

The communication costs showed a predictable shape as suggested by the theoretical analysis in [1]. We compare the theoretical and actual communication costs for the 20-bit universe in the full paper [2]. The algorithm with the

biggest hidden constant is AP with a factor of around 10 attached to the cost.

In the rest of the section we detail our experiments.

4.1 Processing Time, By Universe Size

Each of the datasets used in the experiment was sized to have 2^{20}, 2^{21}, or 2^{22} strings, regardless of the length or number of the strings in the universe, or the size of the universe itself. This allows measuring how the algorithms behave as the number of extraneous strings (and thus the universe complexity) grows.

4.1.1 20-Bit Universe

The total processing, mapper, shuffle (mapper-to-reducer copy), and reducer times for Naive always remain approximately constant (except for slight variations due to cluster node performance).

Ball-Hashing algorithms start out efficiently. However, their shuffle times grow fast, and their mapper times even faster. By a distance threshold of 5, both algorithms have been outsped by Naive. Unlike the other algorithms, the BH family tends towards long mapper times, due to the volume of keys that must be generated and copied. BH2's time grows in steps. This is because the algorithm behaves as if the distance is half its actual value, and therefore its time grows every time the threshold increases by two.

AP's processing time steadily increases; while its mapper and shuffle follow the communication cost graph, its reducer rises with the increased threshold.

Splitting behaves efficiently, with the mapper times approximately equal to the Naive algorithm and shuffle and reducer times lower than every other algorithm. It grows very slowly, taking only a few minutes to process the dataset irrespective of the distance threshold.

4.1.2 24-Bit Universe

In the universe of 24-bit strings, the algorithms now react to the increased complexity.

Naive behaviour for the 1/16 dataset (2^{20} strings) is the same as the 20-Bit universe set, with no difference in time for any phase. Increasing the dataset size to 1/8 (2^{21} strings) or 1/4 (2^{22} strings) does not change Naive's mapper times, but it does increase the reducer time, increasing the time by a factor of 2 to 3 for each doubling of the dataset.

BH algorithms repeat their pattern of growing to enormous heights, here taking longer than Naive by distance 4, for all 24-bit datasets. Increasing the universe size has a drastic effect on this family of algorithms; as an example, the total processing time for the 20-bit dataset at distance 5 is 41 minutes. For the 1/16 24-bit dataset (same number of strings), also at distance 5, it took 1 hour and 50 minutes to process the data. In addition, the communication cost for this family of algorithms grows so large that it becomes impractical to run on larger datasets. For distances of 6 and above (and 5 for the 1/8 dataset), BH1 did not return results in a reasonable amount of time and space.

For the AP algorithm at higher distances, a slight dip is observed at medium thresholds as mapper and shuffle time decrease but reducer time has yet to rise to compensate. This temporarily gives the algorithm better performance than Naive, but worse than Splitting; however, this performance advantage is lost when the dataset becomes large enough (at distance 6 in the 1/4 dataset).

Splitting still netted the lowest processing times overall. Similar to Naive, the mapper and shuffle times of Splitting stay stable.

4.2 Further Observations

In these results, several points of interest emerge:

- For all algorithms, the graphs of the mapper processing time appear to be similar in shape to that of the communication cost; this suggests that the biggest factor on mapper runtime is the time taken to produce the intermediate output. However, as we discuss later (see Disk Utilization), the mappers are not disk-bound.
- Naive was not affected by increasing the string length, even though the same increase was responsible for processing time increases in several of the other algorithms. This suggests that the increased time to compare longer strings is outweighed by the time taken to compare all the strings that arrive at the reducer.
- The time of BH1 grows too quickly to be useful for all but very low distances. BH2 grows in a similar manner, but at double the distances of BH1. Unfortunately, this growth is already quite big to be practical by distance 6 or more.
- For the BH algorithms, the shape of the total processing time appears to be very similar to that of the theoretical (and actual) communication cost. This suggests that the communication cost is a primary influence on the processing time taken for these algorithms.
- Splitting consistently took the least amount of time to complete among all the algorithms.

Compression. We also ran our experiments using the compression option in Hadoop. As an example of compressed results we have included both compressed and uncompressed versions of the results for the 1/8 subset of the 24-bit universe, in Figures 3 and 4. We have included more results with compression in the long version [2]. We observe that using compression cuts the intermediate communication cost by about 75%. However, this caused the Mapper processing time to increase, due to the time necessary to compress the data. Compression had little effect on the reducer processing time, and decreased most shuffle times (with the exception of BH1). Overall, adding compression comes at a tradeoff in total processing time that is approximately proportional to the original size of the intermediate communication cost.

Disk Utilization. The analysis of the disk usage data from the cluster shows that the disks are not becoming saturated. Namely, using the sar utility, we observed that the maximum data rate written to disk at any time during all our experiments was not more than 27.45MB/sec, whereas our disks can handle up to 104MB/sec. This suggests that the mappers and reducers are in fact CPU-bound.

5. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE'12*, pages 498–509, 2012.

[2] B. Kimmett, V. Srinivasan, and A. Thomo. Fuzzy joins in mapreduce: An experimental study (long version), http://webhome.cs.uvic.ca/~thomo/fuzzy.pdf.

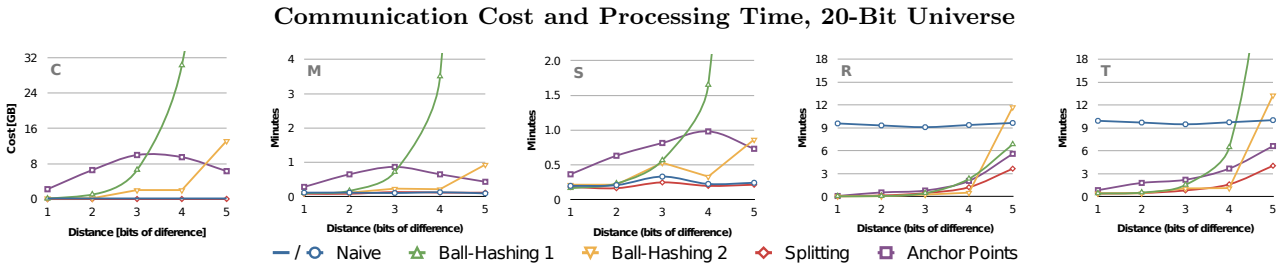

Figure 1: The graphs show the communication cost (C), and the mapper (M), shuffle (S), reducer (R), and total (T) time, respectively, from left to right. (For BH1, distance 5, C, M, S, and T are 84.77 GB, 29.83 min, 11.73 min, and 41.73 min, respectively. They do not fit in the plot area.)

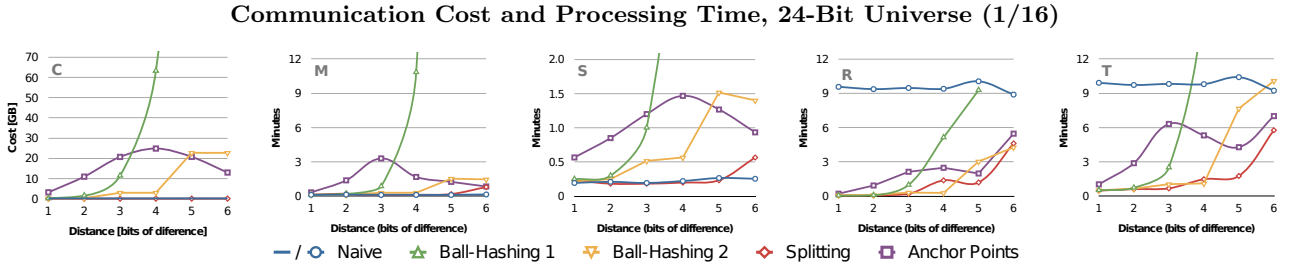

Figure 2: C, M, S, R, and T, respectively, from left to right. (For BH1, C and M at distance 5 are 216.65 GB and 98.87 min, respectively. S at distances 4 and 5 is 5.95 min and 37.3 min, and T at distances 4 and 5 is 18.13 min and 110.35 min. BH1 could not be run for distance 6.) M and R do not sum to T because Hadoop staggers mapper and reducer tasks for efficiency.

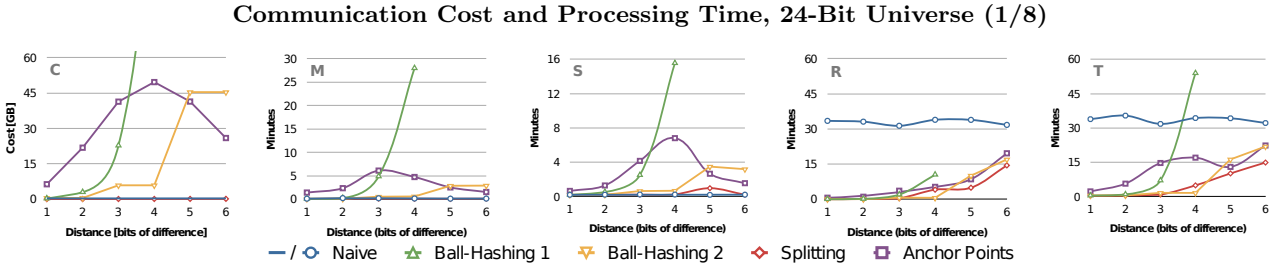

Figure 3: C, M, S, R, and T, respectively, from left to right. (For BH1, distance 4, C is 433.32 GB.) BH1 could not be run for distances 5 and greater.

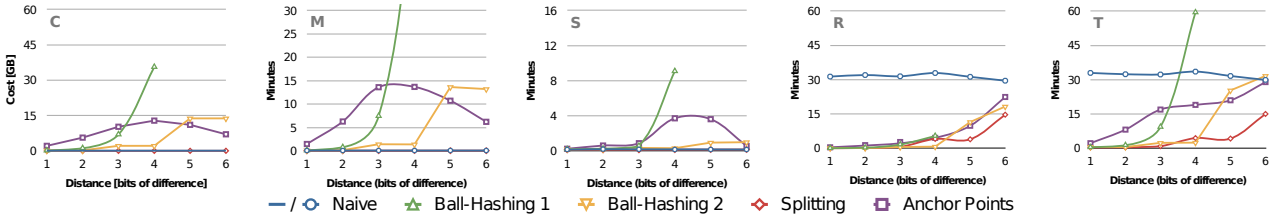

Figure 4: C, M, S, R, and T, respectively, from left to right. (For BH1, distance 4, M is 52.47 min.) BH1 could not be run for distances 5 and greater.