SAND Join — A Skew Handling Join Algorithm for Google's MapReduce Framework

Fariha Atta*†, Stratis D. Viglas†, and Salman Niazi*,
*National University of Computer and Emerging Sciences
Email: {fariha.atta,salman.niazi}@nu.edu.pk
†School of Informatics, University of Edinburgh
Email: fariha.atta@ed-alumni.net, sviglas@inf.ed.ac.uk

***Abstract*—The simplicity and flexibility of the MapReduce framework have motivated programmers of large scale distributed data processing applications to develop their applications using this framework. However, the implementations of this framework, including Hadoop, do not handle skew in the input data effectively. Skew in the input data results in poor load balancing which can swamp the benefits achievable by parallelization of applications on such parallel processing frameworks. The performance of join operation, which is the most expensive and most frequently executed operation, is severely degraded in the presence of heavy skew in the input datasets to be joined. Hadoop's implementation of the join operation cannot effectively handle such skewed joins, attributed to the use of hash partitioning for load distribution. In this work, we introduce "*S*kew h*AND*ling *Join*" (SAND Join) that employs range partitioning instead of hash partitioning for load distribution. Experiments show that SAND Join algorithm can efficiently perform joins on the datasets that are sufficiently skewed. We also compare the performance of this algorithm with that of Hadoop's join algorithms.**

I. Introduction

MapReduce — a software framework developed at Google, provides a cost-effective, scalable, flexible, and fault-tolerant distributed software model to develop applications for large scale distributed data processing across huge clusters of commodity machines. MapReduce facilitates efficient processing of voluminous data in parallel, upto multiples of petabytes [1]. MapReduce takes the responsibility of data distribution, load balancing, fault-tolerance, job scheduling, and other essential details of parallel processing. Users of the MapReduce framework have to concentrate only on data processing algorithms which have to be implemented by users in the *map* and *reduce* functions of the framework.

Map and *reduce* are the two primitives provided by the framework for distributed data processing. Users implement *map* and *reduce* functions to be run on mapper and reducer nodes/machines respectively. The signatures of these primitives for key 'k' and value 'v' are:

$$Map : (k_1, v_1) \rightarrow [(k_2, v_2)]$$

$$Reduce : (k_2, [v_2]) \rightarrow [v_3]$$

Data to be processed is available in key-value pairs. The map function, running on mapper nodes, converts input key-value pairs of data into intermediate key-value pairs which are passed over to reducer nodes for further processing. In simple terms, map phase distributes data among nodes for processing while the output is produced in the reduce phase.

Due to the ease of development and flexibility provided by MapReduce, data intensive applications are migrating towards implementations using the MapReduce framework. Many candidate applications of MapReduce require combining data from multiple sources. However, joining large heterogeneous datasets using MapReduce is an extremely challenging task. This is due to the fact that MapReduce is designed to process a single homogeneous data stream per unit time and hence does not have an efficient description for joining two parallel streams. In Hadoop [2] (an open souce implementation of the MapReduce framework by Apache), some strategies have been documented for the join operation. These are the *Map-side* and *Reduce-side* join techniques. However, these join algorithms have inherent limitations. When it comes to joining skewed datasets, the performance of these join techniques is degraded.

Skew in the distribution of the join attribute's value can overshadow the strengths of parallel processing infrastructure. Skew in the input datasets causes an uneven distribution of load among the parallel sites where the two datasets are joined. The consequent variation in the processing time on parallel sites affects the maximum speedup that can be achieved by virtue of parallel execution. In their popular research article *"Map-Reduce: A major step backwards"* [3], DeWitt and Stonebraker criticize various aspects of MapReduce, one of which is its inability to handle skew. They state: *"One factor that Map/Reduce advocates seem to have overlooked is the issue of skew . . . The problem occurs in the map phase when there is wide variance in the distribution of records with the same key. This variance, in turn, causes some reduce instances to take much longer to run than others, resulting in the execution time for the computation being the running time of the slowest reduce instance. The parallel database community has studied this problem extensively and has developed solutions that the Map/Reduce community might want to adopt"*.

In this work, we present *SAND Join* algorithm for the Hadoop implementation of the MapReduce framework. This algorithm can produce efficient joins when data is highly skewed. We analyze its performance with respect to the implementations provided by Hadoop.

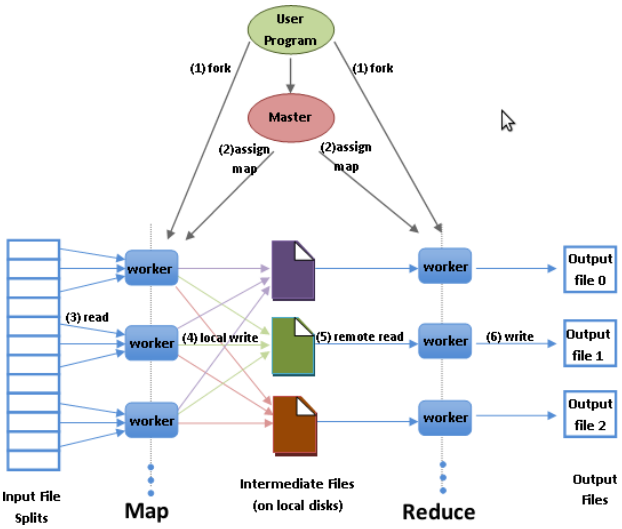

Fig. 1. The HDFS Architecture

II. Overview of Hadoop

The Hadoop Distribution of MapReduce is based on Hadoop Distributed File System (HDFS) [2] — a file-system with master-slave architecture (shown in Fig. 1). In a Hadoop cluster of n nodes, one of the nodes is master node refered as *NameNode (NN)*. Remaining nodes are workers nodes also referred as *DataNodes (DN)*. The NN maintains metadata about the FileSystem. Files are broken down into default split sizes of 64MB and distributed among the DNs. Each split is replicated on three DNs to ensure fault-tolerance. *Jobtrackers* and *tasktrackers* are the daemons residing on NNs and DNs respectively to handle jobs and tasks. When a MapReduce job is submitted to a NN, a jobtracker divides it into m tasks and assigns a task to each mapper. Following is the sequence of steps for conversion of input to output on Hadoop:

1) **Mapping Phase:** Each mapper works on non-overlapping input splits assigned to it by the NN. The map instance, to which the input split is assigned, extracts key and value from each record of the split, applies a user defined map function on the key-value pairs, and produces intermediate key-value pairs. The intermediate results of mappers are written to the local file-system in a sorted order.
2) **Partitioning Phase:** A "partitioner" determines the reducer node to which an intermediate key-value pair should be directed. The default partitioner applies a hash function *(hash_value_of_key) mod (number_of_partitions)* on the key of each key-value pair to allocate a partition number to the key-value pair.
3) **Shuffling Phase:** Each reduce instance is allocated a partition to work on by the NN. The NN informs each reducer about the location of mapper nodes from which it has to copy the data of its partition into its memory. This process of moving data to appropriate reducer-nodes is called "shuffling".
4) **Sorting Phase:** After copying the assigned partition into its memory, each reducer sorts the partition on key-attribute using merge sort algorithm.
5) **Reduce Phase:** Each reduce task applies a user-defined reduce operation (selection, projection, join etc.) on each group of keys and the result is written to HDFS.

III. Related Work

A few join techniques for MapReduce have been discussed in the existing MapReduce literature. Hadoop implements map-side and reduce-side [4], [5] algorithms for joining datasets. As the name suggests, a *reduce-side* join is carried out on reducer nodes. Each mapper node tags every input tuple of both datasets with its source information and generates tagged intermediate key-value pairs. These tagged key-value pairs are shuffled across the network and each reducer node receives the tagged key-value pairs with same join key attribute from both datasets to be joined. The join is carried out by each reducer between the tuples of the two datasets.

In a map-side join, the reduce phase is completely bypassed and the join is carried out in the map phase. Hadoop defines two variants of the map-side join: a) partitioned join b) memory-backed join. A map-side partitioned join can be executed only if the two datasets are already partitioned on the join key and the keys are in sorted order. Each mapper receives two similar partitions, one from each dataset, and computes the join between their tuples. In this case, the shuffling and sorting cost incurred by the reduce-side join can be avoided by carrying out the join in the map phase. As opposed to the partitioned join, a memory-backed join completely copies the smaller of two datasets into the main memory of each mapper where an in-memory hash table is built for this dataset. Each mapper then, on receiving a partition of the second dataset, probes its tuples against the hash-table of first dataset and produces joined records.

A modification to the MapReduce framework for the join operation is presented by Hung-chih et al. [6], called *Map-Reduce-Merge*. It introduces a new stage called "merge" where matching tuples from multiple sources, partitioned and sorted by map and reduce functions, are merged and joined. Although this augmented merge stage makes join operation easier, it can incur an additional overhead for implementation.

The map-side, reduce-side, and map-reduce-merge join algorithms use hash partitioning to distribute datasets among parallel worker nodes. Using hash partitioning scheme, a hash function on the distribution attribute, i.e. the join key, assigns tuples of datasets to parallel sites. However, hash partitioning is sensitive to the presence of skew in the input data i.e. when a significant number of tuples of the dataset has the same value for join attribute. Whenever datasets are distributed among the parallel processing nodes on the basis of hash partitioning function, a key that is skewed will be directed to a single processing node. As a result, the worker nodes handling such tuples are overloaded with data to be joined. Selection of a perfect hash function may restrict two *different* join attribute values to be hashed into the *same* partition. However, even

this ideal hash function will result in a load imbalance since hash partitioning directs same keys to a single partition and hence a partition that receives an excessively used key will be overloaded. Hence, the algorithms using hash partitioning for data distribution are prone to a degraded performance when the input datasets to be joined contain skewed distribution keys. Since all of the Hadoop's join algorithms, as well as the map-reduce-merge extension to Hadoop, use hash-partitioning for data distribution, they suffer from a performance hit in case of skewed data.

To the best of our knowledge, no work has yet been carried out to handle skew in Hadoop's join operation. We present SAND join algorithm for Hadoop that can effectively handle joining of skewed datasets.

IV. The SAND Join Algorithm

The SAND join algorithm is designed to overcome the incapability of existing Hadoop join algorithms to perform skewed joins. A major strength of our methodology is that it can easily be incorporated in the framework without extensive design changing. Instead of using hash partitioning, which directs a skewed key to a single processing node, we use range-based partitioning which distributes the skewed keys among a number of processing nodes. Range-based partitioning exploits the characteristics of data for load balancing; two of such strategies are *simple range partitioning* and *virtual processor range partitioning* [7].

A. Simple Range-based Partitioning

In range-based partitioning, the domain of join keys is divided into a number of blocks, called ranges. The number of ranges is equal to the number of partitions. In the simple range partitioning, the number of partitions is equal to the number of processing units (PUs). In case of a heavy skew, allocating only a sub-range of a single distribution key to one partition reduces the burden on a single PU. A split vector determines the boundaries to distribute values among partitions. Given *p* PUs, the split vector contains *p-1* key entries i.e. $\{k_1, k_2, k_3, \ldots, k_{p-1}\}$. From this split vector, each PU is assigned a lower bound and an upper bound for range partitioning (except the first and last PUs which do not have lower and upper bounds respectivley). All the tuples that have their join key attribute falling in a particular range are sent to the PU associated with that range. For example, $keys \leq k_1$ are routed to PU_1, $k_1 < keys \leq k_2$ are directed to PU_2, and $keys > k_{p-1}$ find their way to PU_p.

Now the question arises that how this split vector should be selected? A good split vector can be selected by sampling the input relations so that an estimated distribution of join attributes in the data can be obtained. Sampling a sorted input dataset is one solution. However, sorting a voluminous dataset takes a significant amount of time. On the other hand, random sampling of the input dataset provides a better estimate of the distribution of join keys in the input dataset without the need to parse the whole input dataset. The random samples not only determine the degree of skew in data, the samples are also used for exploiting parallelism and partitioning the data. The randomly collected samples are stored in a sorted order in a split table *T*. Since the size of this sample table is many folds smaller than the input dataset, sorting *T* does not take significant time. A split vector is determined from this sorted split-table by collecting values after a step size of *size(T)/p*.

Since the input dataset is randomly sampled, there are fair chances that the skewed attribute will occur more than once in the split vector. Let us consider that split vector contains $\{k_1, k_2, k_2, k_3\}$ for a 5-PU distributed system. $Keys \leq k_1$ are assigned to PU_1 but $k_1 < keys \leq k_2$ can be directed to either PU_2 or PU_3. Thus using range partitioning, a skewed join attribute k_2 is distributed among more than one PUs and hence a single machine is not penalized for the skew. For a build relation, if a key can be directed to more than one PUs, a candidate PU is randomly selected and the tuple is sent to it. Whereas for a probe relation, such key is routed to every candidate PU, in order to generate every possible join result.

Implementation of Simple Range Partitioner

As discussed earlier, a *simple range partitioner* partitions the tuples of the relations into a number of ranges on the basis of join key attributes. To accommodate the simple range partitioner in the SAND join algorithm, we replace the default hash partitioner of Hadoop by our range partitioner. The steps to partition a build relation using a simple range partitioner are as follows:

1) In order to sample the input dataset, the NN is directed to collect samples of the join key attributes from each input split before the start of the job. We assume that the join key attributes are randomly distributed over the input datasets and hence over the input splits. For the number of samples *x* specified by user, the NN retrieves *x/y* samples from each of the *y* input splits i.e. equal number of samples from each input split to form a split-table *T*.
2) The split table *T* is sorted. We now retrieve only *p* samples from *T*, where *p* is the number of partitions. This results in a split vector whose entries determine the boundaries for splitting a dataset into a number of range.
3) The split vector is written to a file (we call it a *sampling file*) and stored in HDFS.
4) This sampling file is distributed to all the mapper nodes using the *Distributed Cache* [2] mechanism.
5) Each mapper node uses this sampling file to make partitioning decisions in the partitioning phase. In its *configure()* function (which is called before the start of any map task), each mapper node reads the sampling file from the distributed cache and builds a *range map*. Range-map is a matrix containing information about the ranges and their associated partition numbers. Whenever a mapper reads a (key, value) pair, it checks the range-map to determine the range to which the key belongs. For the matching range, the associated partition number from the range map is allocated to the (key, value) pair. If a join key attribute belongs to more than one ranges,

the mapper randomly assigns to this key one of the candidate partition numbers associated with that range. The partition number is then encoded with the (key, value) pair. When this encoded pair is received by the partitioner, it decodes the value of partition number and returns it to the framework so that the framework can direct the (key, value) pair to an appropriate reducer. *Note: We implement a custom partitioner on the top of* ***Partitioner*** *class to replace the hash partitioning functionality of the default partitioner by our range-based partitioner.*

To partition the probe relation, steps 1-4 are same as for partitioning of build relation. The split vector produced by sampling the build relation is reused for partitioning the probe relation. Unlike the build relation where a key falling into more than one partitions is randomly allocated to a candidate partition, a probe key falling into more than one partitions is allocated the partition number of all the candidate partitions for the key. However, the partitioner class can assign only a single partition number to a key and return this number to the framework. To override the default behavior, we emit such (key, value) pair multiple times (i.e. once per candidate partition), encoding one partition number with each pair. In the partitioner, the partition number is decoded from the encoded (key, value) pair and sent to the MapReduce framework which then instructs an appropriate reducer to pick up the tuple.

B. Virtual Processor Range Partitioner

In the simple range partitioner, each PU handles only one partition and hence the number of partitions is the same as the number of PUs. Range partitioning can be enhanced by making the number of partitions greater than the number of PUs. Skew can be handled efficiently if data is distributed among a greater number of partitions. These partitions can then be assigned to PUs either in a round robin fashion or PUs can dynamically be fed with partitions as soon as they finish their earlier workload. The motivation behind this virtual processor approach is that if data is skewed, having a large number of partitions spreads the skewed data over a greater number of partitions. As a result, work gets more evenly distributed and the system does not suffer from the inefficiencies caused by skew. The number of partitions in the virtual processor approach should be an integer multiple of the number of PUs otherwise it can result in load imbalance.

Considering the example of section IV(A), if we take '2' as a factor for the virtual processor partitioning, our split vector will contain 10 instead of 5 splits. The split vector for this case may have key entries $\{k_1, k_2, k_2, k_2, k_3, k_4, k_5, k_6, k_7\}$. A key falling in a particular range is allocated to its associated partition e.g. keys $\leq k_1$ are assigned to partition 1; $k_1 < keys \leq k_2$ are assigned to partition 2, 3, or 4; $k_2 < keys \leq k_3$ are allocated to partition 5, and so on. As evident, join keys k_2 are dispersed across three partitions instead of two. This reduces the accumulation of the skewed key k_2 in a single partition.

Implementation of Virtual Processor Range Partitioner A virtual processor range partitioner (VPR) is implemented in the same way as the simple range partitioner except that the number of partitions in VPR is a multiple of the number of *reduce* nodes. Hadoop schedules the partitions itself on the reduce nodes. Initially, it allocates a single partition to each node. When a node completes the join of one pair of partitions, the NN allocates a partition from the pool of pending partitions to the idle reduce node. This enables dynamic load balancing and hence skew is handled in an effective way.

C. Implementation of SAND Join Algorithm

After replacing the hash partitioner with our range partitioner, the join between two datasets is computed. We provide two versions of the SAND join algorithm, one implemented using simple-range partitioner and the other using virtual processor range partitioner. We perform grace hash-joining [8] on the datasets to be joined. Following is the sequence of steps for SAND join algorithm implementation for the join of R and S (shown in Fig. 2):

- **Step 1:** Lets assume *S* is smaller of the two relations and *S* is not already partitioned into *x* partitions on the basis of the join key. We run a MapReduce job to partition it with either simple or virtual processor partitioner. Remember that we have replaced default hash partitioner with range partitioner.
- **Step 2:** In a second MapReduce job, we read the input splits of the outer relation *R* and pass them through an identity mapper because we are not required to do anything in the mapper. We want our data to pass through the partitioning stage. The partitioning function should generate *x* partitions on the join key so that each partition of *R* can have a corresponding partition of *S* containing the same keys. For this purpose, the partitioning function for both relations should be the same. Our custom range partitioner here partitions the relations on the basis of ranges of keys.
- **Step 3:** After the partitioning stage, each reducer receives one partition of the outer relation *R*. This reducer now needs the corresponding partition of the relation *S* which is stored in HDFS. For each reducer, we determine the partition number of relation *R* it is dealing with by using the *mapred.task.partition* property of the *JobConf* object. This property returns an integer representing the partition number being handled by the reducer. Using this partition number, an HDFS path is constructed for the corresponding partition of the relation *S* and the partition from HDFS is loaded.
- **Step 4:** An in-memory hash-table is built for the loaded S partition. We want this hash table to be built before any reduce task is carried out (where we perform the actual join). Loading of the partition and making of the hash table is therefore done in the *configure()* function.
- **Step 5:** As stated earlier, due to the same partitioning functions applied, each reducer handles similar partitions form relations *R* and *S*. From its assigned partition of relation R, each reducer provides the keys and their associated sets of values to the corresponding reduce

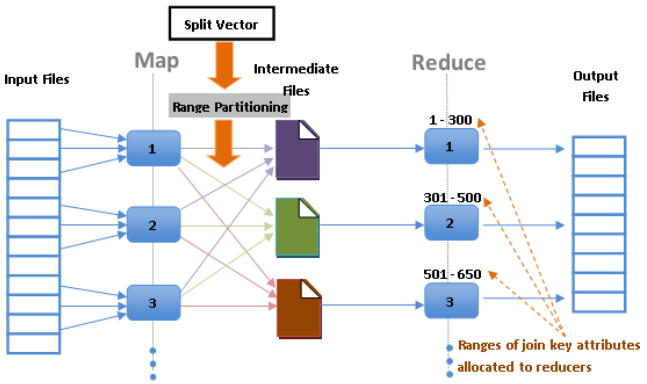

Fig. 2. Custom Range Partitioning for Hadoop

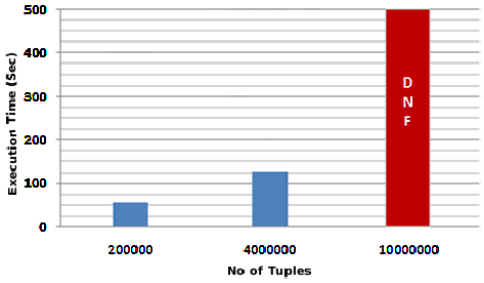

Fig. 3. Time taken by the memory-backed join for varying number of tuples in the build relation (*DNF=Does Not Finish)

tasks. In each *reduce* task, the key is probed against the hash-table. If one or more matches are found in the hash-table, a join is computed with all the associated values and the result of the join is written to HDFS.

V. Evaluation

In this section, we experimentally evaluate the performance of the join algorithms discussed above for handling different degrees of skew. To compare the performance, we consider the absolute runtimes of the algorithms.

A. The Testbed

For the purpose of experimentation, we use a Hadoop cluster of eight nodes. Out of these eight nodes, six are DNs, one is NN responsible for managing the distributed file system, and one is tasktracker node responsible to assign the map and reduce tasks to DNs. Each node is a Dell Poweredge SC1425 with two Intel Xeon 3.2 GHz CPUs and a 256MB/16GB ECC DDR-2 400 SDRAM memory chip. The secondary storage of each node is 80GB SATA drive running at 7200rpm. The nodes are connected to an HP ProCurve 2650 at a network bandwidth of 100BaseTx-FD. The cluster contains two racks, each consisting of three DNs. The racks are connected by a 1Gbps link. On each node Scientific Linux 5.5, Hadoop 0.18, and Java 1.6 are installed. The block size is the default 64MB. The size of heap memory is increased to 1024MB. Two map and two reduce tasks can run on each node, providing a total capacity of 4 tasks per node.

B. Datasets

We generate datasets such that join key attributes are distributed randomly over each dataset. For the sake of performance comparison, we construct datasets of cardinalities 2,000,000 with varying degrees of skew. To keep things simple, we take join key attribute as simple numerical values, ranging from *1 - 2,000,000*. In all skewed datasets, the skewed join key has the attribute value "1". The convention used to represent the datasets is like this: *d1* represents an input dataset that has only a single 1 as the join key. All the other join key attributes are randomly assigned values from 2 to 2,000,000. Similarly, *d10* represents an input dataset with ten 1s and the remaining 1,999,990 values are randomly assigned values from 2 to 2,000,000; *d100* has 100 1s; *d1K* has 1000 1s; *d20K* has 20,000 1s, and so on. We represent a join as, for example, *d1* x *d1K* to show a join between two input datasets, one containing a single join attribute with value 1 and the other consisting of 1000 join attributes having value 1. When we represent a join as *d1* x *d10*, the first of these datasets is the build relation and the second is the probe relation. Each tuple in the datasets consists of a join key, two random date values, and two random strings. The average size of a tuple is 80 bytes; the join key occupies 8 bytes. Following are the conventions used for representing the algorithms.

RJ = Reduce-Side Join
MPJ = Map Side Partitioned Join
MBJ = Memory Backed Join
SAND-R = SAND Join with Simple Range Partitioning
SAND-VP = SAND Join with Virtual Processor Range Partitioning

C. Tests

We conduct experiments to analyze the performance of Hadoop joins (i.e. map-side partitioned, memory-backed, and reduce-side joins) and the simple range-based and virtual processor based SAND join algorithm.

We first carry out an experiment to show a limitation of memory-backed joins (MBJ). In Fig. 3, we present the results of the join operation using the memory-backed algorithm for the build relation of varying number of tuples, keeping the number of tuples in the probe relation constant. When the size of build relation gets large (i.e. 10,000,000 records in Fig. 3), the join operation never comes to an end. This is because the nodes computing the join run out of the memory while building an in-memory hash table for such a large dataset. Therefore, in general, the memory-backed join is not suitable for computing joins of huge datasets. Due to this limitation of MBJ algorithm, we do not further consider it in experimentation.

We now present the performance comparison of the reduce-side and map-side partitioned join algorithms against SAND join algorithm while varying the degree of skew in the input data. The number of partitions for RJ and MPJ is taken to be 12. The SAND-R join algorithm has same number of partitions as the number of nodes i.e. 6. For the virtual processor range

TABLE I
EXECUTION TIMES FOR JOIN OPERATION WITH SKEW IN BUILD RELATION

Algo.	$d1$ x $d1$	$d10K$ x $d1$	$d100K$ x $d1$	$d300K$ x $d1$	$d400K$ x $d1$	$d500K$ x $d1$	$d600K$ x $d1$
RJ	157	121	98	93	133	135	147
MPJ	193	157	119	113	167	175	180
SAND-R	190	163	127	114	158	153	146
SAND-VP	180	159	123	105	148	133	137

TABLE II
EXECUTION TIMES FOR JOIN OPERATIONS WITH SKEW IN BUILD AND PROBE RELATIONS

Algo.	***d*300K x *d*10**	***d*500K x *d*10**	***d*100K x *d*100**	***d*300K x *d*100**
RJ	76	243	350	927
MPJ	69	256	368	896
SAND-R	66	222	359	918
SAND-VP	53	179	340	718

partitioning in the SAND-VP join algorithm, we keep the factor of virtual processor '2' i.e. the total number of partitions is 12. We conduct each experiment three times and present here the mean of those values. The results of join with datasets of varying skew are presented in Table I and Table II. Table I shows the results of join in case when only the build relation is skewed whereas joins in Table II are computed when both the build and probe relations are skewed.

The results in the tables make it clear that the reduce-side join performs better than other algorithms in little- or no- skew situations. This is because all other algorithms require partitioning of datasets on the join key to accumulate similar keys of both datasets into partitions with same partition number. The two corresponding partitions of both datasets are then joined together in the join stage. Since the reduce-side join skips this phase of partitioning, its performance is better than the other two algorithms. However as the skew increases, the performance of the reduce-side join starts degrading. The reason is that the keys are distributed among the reducer nodes (where join takes place) according to the hash code of the join key. The reduce tasks receiving the skewed keys are overloaded as compared to the other reduce tasks and hence the overloaded tasks take more time to compute the join, thereby degrading the performance, as evident from Table II.

It is clear from Table II that in case of heavy skew, SAND join algorithm has better performance than reduce-side and map-side partitioned join. It is also clear that the virtual processor partitioning approach (SAND-VP) performs the best among all algorithms because it distributes skewed keys across a greater number of partitions. The load imbalance is prevented and the system does not have to wait for the heavy-hitter nodes to complete the processing. For heavily skewed joins, the simple range partitioning version (which uses the same number of partitions as the number of nodes) although performs better than algorithms using the hash partitioning, its performance is always lower than that of the virtual processor partitioning approach. This is because of the fact that the skewed keys are distributed to a greater number of partitions by the virtual processor partitioning as compared to the simple range partitioning approach. As evident from Table I, in case of little or no-skew, hash-partitioned algorithms (in particular, RJ algorithm) have better efficiency than range-partitioned algorithms. For this reason, we modify our algorithm such that it detects the degree of skew in the samples of input datasets and dynamically selects an appropriate partitioning strategy i.e. hash-partitioning in case of little skew and virtual processor range partitioning strategy for heavily skewed joins.

VI. CONCLUSION

Parallel joins are vulnerable to skew in datasets being joined. If datasets are sufficiently skewed on some keys, load imbalances may result. The hash partitioning strategy is found to be inefficient for load distribution in case of skewed data. Instead, if range partitioning is employed, the datasets are distributed among processing nodes on the basis of the characteristics of data itself. The Hadoop algorithms of join operation do not efficiently handle the joining of sufficiently skewed datasets since they employ the hash partitioning approach. In this paper, we presented SAND Join algorithm that is capable of handling skew in the input datasets by virtue of range partitioning. Since hash partitioning performs well in case of non-skewed data and range partitioning has a better performance when the input data is skewed, our algorithm dynamically selects an appropriate partitioning strategy on the basis of the characteristics of input data, determined through random sampling of data. In case of non-skewed data, hash partitioning strategy is selected while if significant skew in the input data is detected, our algorithm selects range partitioning strategy (i.e. virtual processor approach) for the partitioning phase.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *ACM Commun.*, vol. 51, no. 1, pp. 107–113, Jan 2008.

[2] Apache hadoop. [Online]. Available: http://hadoop.apache.org/common/docs/current/mapred_tutorial.html

[3] D. DeWitt and M. Stonebraker, "Mapreduce: A major step backwards blogpost," http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[4] T. White, *Hadoop: The Definitive Guide*, first edition ed., M. Loukides, Ed. O'Reilly, june 2009.

[5] C. Lam, *Hadoop in Action*. Manning Publications, 12 2010. [Online]. Available: http://amazon.de/o/ASIN/1935182196/

[6] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of international Conference on Management of Data*, 2007, pp. 1029–1040.

[7] S. Ghandeharizadeh and D. J. DeWitt, "Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines," in *Proceedings of 16th International Conference on Very Large Data Bases*, 1990, pp. 481–492.

[8] Grace hash join. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa178403%28v=sql.80%29.aspx