Generalized Parallel Join Algorithms and Designing Cost Models

Alice Pigul

SPbSU
m05pay@math.spbu.ru

Abstract

Applications for large-scale data analysis use such techniques as parallel DBMS, MapReduce (MR) paradigm, and columnar storage. In this paper we focus in a MapReduce environment. The aim of this work is to compare the different join algorithms and designing cost models for further use in the query optimizer.

1 Introduction

Data-intensive applications include large-scale data warehouse systems, cloud computing, data-intensive analysis. These applications have their own specific computational workload. For example, analytic systems produce relatively rare updates but heavy select operation with millions of records to be processed, often with aggregations.

There are the following architectures that are used to analyze massive amounts of data: MapReduce paradigm, parallel DBMSs, column-wise store, and various combinations of these approaches.

Applications of this type process multiple data sets. This implies need to perform several join operation. It's known join operation is one of the most expensive operations in terms both I / O and CPU costs.

Unfortunately, join algorithms is not directly supported in MapReduce. There are some approaches to solve this problem by using a high-level language PigLatin, HiveQL for SQL queries or implementing algorithms from research papers. The aim of this work is to generalize and compare existing equi-join algorithms with some optimization techniques and build cost model which could be used in a query optimizer for a distributed DBMS with MapReduce.

This paper is organized as follows the section 2 describe state of the art. Join algorithms and some optimization techniques were introduced in 3 section. The designing of cost models for join algorithms are presented in 4 section. Performance evaluation will be described in 5 section. Finally, future direction and some discussion of experiments will be given.

Proceedings of the Spring Young Researcher's Colloquium On Database and Information Systems SYRCoDIS, Moscow, Russia, 2012

2 Related work

2.1 Architectural Approaches

Column storage is one of the architectural approaches to store data in columns, that the values of one field are stored physically together in a compact storage area. Column storage strategy improves performance by reducing the amount of unnecessary data from disk by excluding the columns that are not needed. Additional gains may be obtained using data compression. Storage method in columns outperforms row-based storage for workloads typical for analytical applications, which are characterized by heavy selection operation from millions of records, often with aggregation and by infrequent update operation. For this class of workloads I/O is major factor limited the performance. Comparison of column-wise and row-wise stores approaches is presented in [1].

Another architectural approach is a software framework MapReduce. Paradigm MapReduce was introduced in [11] to process massive amounts of unstructured data.

Originally, this approach was contrasted with a parallel DBMS. Deep analysis of the advantages and disadvantages of these two architectures was presented in [25,10].

Later, hybrid systems appeared in [9, 2]. There are three ways to combine approaches MapReduce and parallel DBMS.

- MapReduce inside a parallel DBMS. The main intention is to move computation closer to data. This architecture can be exemplified with hybrid database Greenplum with MAD approach [9].
- DBMS inside MapReduce. The basic idea is to connect multiple single node database systems using MapReduce as the task coordinator and network communication layer. An example is a hybrid database HadoopDB [2].
- MapReduce aside of the parallel DBMS. MapReduce is used to implement an ETL produced data to be stored in parallel DBMS.

This approach is discussed in [28] Vertica, which also supports the column-wise store.

Another group of hybrid systems combines MapReduce with column-wise store. MapReduce and column-wise store are effective in data-intensive applications. Hybrid systems based on this two techniques may be found in [20,13].

2.2 Algorithms for Join Operation

Detailed comparison of relational join algorithms was presented in [26]. In our paper, the consideration is restricted to a comparison of joins in the context of MapReduce paradigm.

Papers which discuss equi-join algorithms can be divided into two categories which describe join algorithms and multi join execution plans.
The former category deals with design and analyses join algorithm of two data sets. A comparative analysis of two-way join techniques is presented in [6, 4, 21]. The cost model for two-way join algorithms in terms of cost I/O is presented in [7, 17].

The basic idea of multi-way join is to find strategies to combine the natural join of several relations. Different join algorithms from relation algebra are presented in [30]. The authors introduce the extension of MapReduce to facilitate implement relation operations. Several optimizations for multi-way join are described in [3, 18]. Authors introduced a one-to-many shuffling strategy. Multi-way join optimization for column-wise store is considered in [20, 32].

Theta-Joins and set-similarity joins using MapReduce are addressed in [23] and [27] respectively.

2.3 Optimization techniques and cost models

In contrast to the sql queries in parallel database, the MapReduce program contains user-defined map and reduce functions. Map and reduce functions can be considered as a black-box, when nothing is known about these functions, or they can be written on sql-like languages, such as HiveQL, PigLatin, MRQL, or sql operations can be extracted from functions on semantic basis. Automatic finding good configuration settings for arbitrary program offered in [16]. Theoretical designing cost models for arbitrary MR program for each phase separately presented in [15]. If the MR program is similar to the semantics of SQL, it allows us to construct a more accurate cost model or adapt some of the optimization techniques from relational databases. HadoopToSQL [22] allows to take advantage of two different data storages such as SQL database and the text format in MapReduce storage and to use index at right time by transforming the MR program to SQL. Manimal system [17] uses static analysis for detection and exploiting selection, projection and data compression in MR programs and if needed to employ B+ tree index.
New SQL-like query language and algebra is presented in [12]. But they are needed cost model based on statistic. Detailed construction of the model to estimate the I/O cost for each phase separately is given in [24]. Simple theoretical considerations for selecting a particular join algorithm are presented in [21]. Another approach [7] for selecting join algorithm is to measure the correlation between the input size and the join algorithm execution time with fixed cluster configuration settings.

3 Join algorithms and optimization techniques

In this section we consider various techniques of two-way joins in MapReduce framework. Join algorithms can be divided into two groups: Reduce-side join and Map-side join. The pseudo code presented in Listings, where R – right dataset, L – left dataset, V – line from file, Key – join key, that was parsed from a tuple, in this context tuple is V.

3.1 Reduce-Side join

Reduce-side join is an algorithm which performs data pre-processing in Map phase, and direct join is done during the Reduce phase. Join of this type is the most general without any restriction on the data. Reduce-side join is the most time-consuming, because it contains an additional phase and transmits data over the network from one phase to another. In addition, the algorithm has to pass information about source of data through the network. The main objective of the improvement is to reduce the data transmission over the network from the Map task to the Reduce task by filtering the original data through semi-joins. Another disadvantage of this class of algorithms is the sensitivity to the data skew, which can be addressed by replacing the default hash partitioner with a range partitioner.
There are three algorithms in this group:

- General reducer-side join,
- Optimized reducer-side join,
- the Hybrid Hadoop join.

General reducer-side join is the simplest one. The same algorithms are called Standard Repartition Join in [6]. The abbreviation is GRSJ.

```
Map (K: null, V from R or L)
    Tag = bit from name of R or L;
    emit (Key, pair(V,Tag));

Reduce (K': join key, LV: list of V with key K')
    create buffers $B_r$ and $B_l$ for R and L;
    for t in LV do
        add t.v to $B_r$ or $B_l$ by t.Tag;
    for r in $B_r$ do
        for l in $B_l$ do
            emit (null, tuple(r.V,l.V));
```

Listing 1: GRSJ.

This algorithm has both Map and Reduce phases. In the Map phase, data are read from two sources and tags are attached to the value to identify the source of a key/value pair. As the key is not effecting by this tagging, so we can use the standard hash partitioner. In Reduce phase, data with the same key and different tags are joined with nested-loop algorithm. The problems of this approach are that the reducer should have sufficient memory for all records with a same key; and the algorithm sensitivity to the data skew.

Optimized reducer-side join enhances previous algorithm by overriding sorting and grouping by the key, as well as tagging data source. Also known as Improved Repartition Join in [6], Default join in [14]. The abbreviation is ORSJ. In the algorithm all the values of the first tag are followed by the values of the second one. In contrast with the General reducer-side join, the tag is attached to both a key and a value. Due to the fact that the tag is attached to a key, the partitioner must be overridden in order to split the nodes by the key only. This case requires buffering for only one of input sets. Optimized reducer-side join inherits major disadvantages of General reducer-side join namely the transferring through the network additional information about the source and the algorithm sensitivity to the data skew.

```
Map (K:null, V from R or L)
    Tag = bit from name of R or L;
    emit (pair(Key,Tag), pair(V,Tag));

Partitioner(K:key, V:value, P:the number of reducers)
    return hash_f(K.Key) mod P;

Reduce (K': join key, LV: list of V' with key K')
    create buffers $B_r$ for R;
    for t in LV with t.Tag corresponds to R do
        add t.v to  $B_r$;
    for l in LV with l.Tag corresponds to L do
        for r in $B_r$ do
            emit (null, tuple(r.V,l.V));
```

Listing 2: ORSJ.

The Hybrid join [4] combines the Map-side and Reduce-side joins. The abbreviation is HYB.

```
Job 1: partition the smaller file S
  Map (K:null, V from S)
      emit (Key,V);

  Reduce (K':join key, LV: list of V' with key K')
      for t in LV do
          emit (null, t);

Job 2: join two datasets
  Map (K:null, V from B)
      emit (Key,V);

  init()  //for Reduce phase
      read needed partition of output file from Job 1;
      add it to hashMap(Key, list(V)) H;
  Reduce (K':join key, LV: list of V' with key K')
      if(K' in H) then
          for r in LV do
              for l in H.get(K') do
                  emit (null, tuple(r,l));
```

Listing 3: HYB.

In Map phase, we process only one set and the second set is partitioned in advance. The pre-partitioned set is pulled out of blocks from a distributed system in the Reduce phase, where it is joined with another data set that came from the Map phase. The similarity with the Map-side join is the restriction that one of the sets has to be split in advance with the same partitioner, which will split the second set. Unlike Map-side join, it is necessary to split in advance only one set. The similarity with the Reduce-side join is that algorithm requires two phases, one of them for pre-processing of data and one for direct join. In contrast with the Reduce-side join we do not need additional information about the source of data, as they come to the Reducer at a time.

3.2 Map-Side join

Map-side join is an algorithm without Reduce phase. This kind of join can be divided into two groups. First of them is partition join, when data previously partitioned into the same number of parts with the same partitioner. The relevant parts will be joined during the Map phase. This map-side join is sensitive to the data skew. The second is in memory join, when the smaller dataset send whole to all mappers and bigger dataset is partitioned over the mappers. The problem with this type of join occurs when the smaller of the sets can not fit in memory.

There are three methods to avoid this problem:

- JDBM-based map join,
- Multi-phase map join,
- Reversed map join.

Map-side partition join algorithm assumes that the two sets of data pre-partitioned into the same number of splits by the same partitioner. Also known as default map join. The abbreviation is MSPJ. At the Map phase one of the sets is read and loaded into the hash table, then two sets are joined by the hash table. This algorithm buffers all records with the same keys in memory, as is the case with skew data may fail due to lack of enough memory.

```
Job 1: partition dataset S as in HYB
Job 2: partition dataset B as in HYB
Job 3: join two datasets
  init()   //for Map phase
      read needed partition of output file from Job 1;
      add it to hashMap(Key, list(V)) H;
  Map(K:null, V from B)
      if (K in H) then
          for r in LV do
              for l in H.get(K) do
             emit(null, tuple(r,l));
```

Listing 4: MSPJ.

Map-side partition merge join is an improvement of the previous version of the join. The abbreviation is MSPMJ. If data sets in addition to their partition are sorted by the same ordering, we apply merge join. The advantage of this approach is that the reading of the second set is on-demand, but not completely, thus memory overflow can be avoided. As in the previous cases, for optimization can be used the semi-join filtering and range partitioner.

```
Job 1: partition S dataset as in HYB
Job 2: partition B dataset as in HYB
Job 3: join two datasets
  init() //for Map phase
    find needed partition SP of output file from Job 1;
    read first lines with the same key K2 from SP and add
        to buffer B;
  Map(K:null, V from B)
     while (K > K2) do
          read T from SP with key K2;
          while (K == K2) do
              add T to B;
              read T from SP with key K2;
     if (K == K2) then
          for r in B do
             emit(null, tuple(r,V));
```
Listing 5: MSPMJ.

In-Memory Join does not require to distribute original data in advance unlike the versions of map joins discussed above. The same algorithms are called Map-side replication join in [7], Broadcast Join in [6], Memory-backed joins [4], Fragment-Replicate join in [14]. The abbreviation is IMMJ. Nevertheless, this algorithm has a strong restriction on the size of one of the sets: it must fit completely in memory. The advantage of this approach is its resistance to the data skew because it sequentially reads the same number of tuples at each node. There are two options for transferring the smaller of the sets:

- using a distributed cache,
- reading from a distributed file system.

```
init() // for Map phase
  read S from HDFS;
  add it to hashMap(Key, list(V)) H;
map (K:null, V from B)
   if (K in H) then
     for l in H.get(K) do
         emit (null, tuple(v,l));
```
Listing 5: IMMJ.

The next three algorithms optimize the In-Memory Join for a case, when two sets are large and no of them fits into the memory.

JDBM-based map join is presented in [21]. In this case, JDBM library automatically swaps hash table from memory to disk.

```
The same as IMMJ, but H is implemented by HTree
instead of hashMap .
```
Listing 6: JDBM.

Multi-phase map join [21] is algorithm where the smaller of the sets is partitioned into parts that fit into memory, and for each part runs In-Memory join. The problem with this approach is that it has a poor performance. If the size of the set, which to be put in the memory is increased twice, the execution time of this join is also doubled. It is important to note that the set, which will not be loaded into memory, will be read many times from the disk.

```
For part P from S that fit into memory do IMMJ(P,B).
```
Listing 7: Multi-phase map join.

Idea of Reversed map join [21] approach is that the bigger of the sets, which is partitions during the Map phase, loading in the hash table. Also known as Broadcast Join in [6]. The abbreviation is REV. The second dataset is read from a file line by line and joined using a hash table.

```
init()  //for Map phase
  read S from HDFS;
  add it to hashMap(Key, list(V)) H;
map (K:null, V from S)
   add to hashMap(Key, V) H;
close()  //for Map phase
    find B in HDFS
    while (not end B) do
        read line T;
        K = join key from tuple T;
        if (K in H) then
             for l in H.get(K) do
                 emit(null, tuple(T,l));
```
Listing 7: REV.

3.3 Semi-Join

Sometimes a large portion of the data set does not take part in the join. Deleting of tuples that will not be used in join significantly reduces the amount of data transferred over the network and the size of the dataset for the join. This preprocessing can be carried out using semi-joins by selection or by a bitwise filter. However, these filtering techniques introduce some cost (an additional MR job), so the semi-join can improve the performance of the system only if the join key has low selectivity. There are three ways to implement the semi-join operation:

- a semi-join using bloom-filter,
- semi-join using selection,
- an adaptive semi-join.

Bloom-filter is a bit array that defines a membership of element in the set. False positive answers are possible, but there are no false-negative responses in the solution of the containment problem. The accuracy of the containment problem solution depends on the size of the bitmap and on the number of elements in the set. These parameters are set by the user. It is known that for a bitmap of fixed size *m* and for the data set of n tuples, the optimal number of hash functions is $k=0.6931*m/n$. In the context of MapReduce, the semi-join is performed in two jobs. The first job consists of the Map phase, in which keys from one set are selected and added to the Bloom-filter. The Reduce phase combines several Bloom-filters from first phase into one. The second job consists only of the Map phase, which filters the second data set with a Bloom-filter constructed in previous job. The accuracy of this approach can be improved by increasing the size of the bitmap. However in this case, a larger bitmap consumes more amounts of memory. The advantage of this method is its the compactness. The performance of the semi-join using Bloom-filter highly depends on the balance between the Bloom-filter size, which increases the time needed for its reconstruction of the filter in the second job, and the number of false positive responses in the containment solution. The large size of the data set can seriously degrade the performance of the join.

```
Job 1: construct Bloom filter
    Map (K:null, V from L)
        Add Key to BloomFilter Bl
    close()  //for Map phase
        emit(null, Bl);

    Reduce (K': key, LV) //only 1 Reducer
      for l in LV do
        union filters by operation Or
      close()  // for Reduce phase
        write resulting filter into file;

Job 2: filter dataset
    init()   //for Map phase
        read filter from file in Bl
    Map (K:null, V from R)
        if (Key in Bl)  then
          emit (null, V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

Listing 7: Semi-join using Bloom-filter.

Semi-join with selection extracts unique keys and constructs a hash table. The second set is filtered by the hash table constructed in the previous step. In the context of MapReduce, the semi-join is performed in two jobs. Unique keys are selected during the Map phase of the first job and then they are combined into one file during the Map phase. The second job consists of only the Map phase, which filters out the second set. The semi-join using selection has some limitations. Hash table in memory, based on records of unique keys, can be very large, and depends on the key size and the number of different keys.

```
Job 1: find unique keys
    Map (K:null, V from L)
        Create HashMap H;
        if (not Key in H) then
            add Key to H;
            emit (Key, null);

    Reduce (K': key, LV) //only one Reducer
        emit (null,key);

Job 2: filter dataset
    init()  //for Map phase
        add to  HashMap H unique keys from job 1;
    Map (K:null, V from R)
        if (Key in H) then
          emit (null,V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

Listing 8: Semi-join with selection.

The Adaptive semijoin is performed in one job, but filters the original data on the flight during the join. Similar to the Reduce-side join at the Map phase the keys from two data sets are read and values are set equal to tags which identify the source of the keys. At the Reduce phase keys with different tags are selected. The disadvantage of this approach is that additional information about the source of data is transmitted over the network.

```
Job 1: find keys which are present in two datasets
    Map (K:null, V from R or L)
        Tag = bit from name of R or L;
        emit (Key,Tag);

    Reduce (K': join key, LV: list of V with key K')
        Val = first value from LV;
        for t in LV do
            if (not Val==Val2) then
              emit (null, K');

Job 2: before joining it is necessary to filter the smaller
dataset by keys from the Job 1 that will be loaded into
hash map. Then the bigger dataset is joined with filtered
one.
```

Listing 8: Adaptive semi-join.

3.4 Range Partitioners

All algorithms, except the In-Memory join and their optimizations are sensitive to the data skew. This section describes two techniques of the default hash partitioner replacement.

A Simple Range-based Partitioner [4] (this kind similar to the Skew join in [14]) applies a range vector of dimension *n* constructed from the join keys before starting a MR job. By this vector join keys will be splitted into *n* parts, where *n* is the number of Reduce jobs. Ideally partitioner vector is constructed from the whole original set of keys, in practice a certain number of keys is chosen randomly from the data set. It is known that the optimal number of keys for the vector construction is equal to the square root of the total number of tuples. With a heavy data skew into a single key value, some elements of the vector may be identical. If the key belongs to multiple nodes, a node is selected randomly in the case of data on which to build a hash table, otherwise the key is sent to all nodes (to save memory as a hash table is contained in the memory).

Virtual Processor Partitioner [4] is an improvement of the previous algorithm based on increasing the number of partition. The number of parts is specified multiple of the tasks number. The approach tends to load the nodes with the same keys uniformly (compared with the previous version). The same keys are scattered on more nodes than in the previous case.

```
//before the MR job starts
// optimal max = sqrt(|R|+|L|)
getSamples (Red:the number of reducers, max: the max
            number of samples)
    C = max/Splits.length;
    Create buffer B;
    for s in Splits of R and L do
        get C keys from s;
        add it to B;
    sort B;
    //in case simple range partitioner P == 1
    //in case virtual  range partitioner P > 1
    for j<(Red*P) do
        T = B.length/(Red*P)*(j+1);
        write into file B[T];

Map(K:null, V from L or R)
    Tag = bit from name of R or L;
    read file with samples and add samples to Buffer B;
    //in case virtual partition it is needed to
    // each index mod |Reducers|
    Ind = {i:  B[i-1] < Key <= B[i]}
    // Ind may be array of indexes in skew case
    if (Ind.length >1) then
        if (V in L) then
            node = random(Ind);
            emit (pair(Key, node), pair(V, Tag));
        else
            for i in Ind do
                emit (pair(Key, i), pair(V, Tag));
    else
        emit (pair(Key, Ind), pair(V, Tag));

Partitioner (K:key, V:value, P:the number of reducers)
    return K.Ind;

Reducer (K': join key, LV: list of V' with key K')
    The same as GRSJ
```
Listing 8: The range partitioners.

3.5 Distributed cache

The advantage of using distributed cache is that data set are copied only once at the node. It is especially effective if several tasks at one node need the same file. In contrast the access to the global file system needs more communication between the nodes. Better performance of the joins without the cache can be achieved by increasing number of the files replication, so there's a good chance to access the file version locally.

4 Cost model

Due to significant differences between parallel DBMS and MapReduce, the MapReduce paradigm requires another optimization techniques based on indexing and compression, programming models, data distribution and query execution strategy. Therefore, we need a different strategy of designing model cost. There are two types of designing cost models: the task execution simulation [29] and analytical cost calculation [15, 24]. To measure the query parallelism effectiveness, it is need to build a cost model that can describe the behavior of each algorithm for parallel query. Analytical model is cost formulas that are used to calculate the query execution time, taking into account the specific of parallel algorithm. Below, analytical cost model for join algorithms and their optimizations will be constructed.

4.1Configuration settings

Execution of MR program depends on input data statistic such as selectivity, skew, compression, on cluster resource such as number of nodes, on configuration parameters, such as I/O cost, and on properties of specific algorithm. Below, the parameters used in the analysis are presented in table.

Variable	Description
s(x)	Size of x in mb
p(x)	Number of pairs for split x
wid	Pair width
c_t	The average computation time needed per pair
pC	The cost for partition
sC	The cost for serialization
sortC	The cost for sorting on keys
cC	The cost for executing combine function
mC	The cost for merge
selP	Selectivity of pairs
selC	Selectivity of combining
|red|	Number of reducers
|map|	Number of mappers
r_h	The cost for reading from HDFS
w_h	The cost for writing to HDFS
rw_l	The cost for local I/O operations
tC	The cost of network transfer
sortMB	io.sort.mb parameter in Hadoop configuration
sortRP	io.sort.record.percent
sortSP	io.sort.spill.percent
F	io.sort.factor
shuBP	mapred.job.shuffle.input.buffer.percent
shuMP	mapred.job.shuffle.merge.percent
memMT	mapred.inmem.merge.threshold
memT	mapred.child.java.opts
redBP	mapred.job.reduce.input.buffer.percent

4.2 Cost of arbitrary MR program

As mentioned above, the MR job consists of the execution stages, thus it is possible to estimate each phase separately. Job may contain the following stages: Setup, Read (read map input), Map (map function), Buffer (serializing to buffer, partitioning, sorting, combining, compressing, write output data to local disk), Merge (merging spill files), Shuffle (transferring map output to reducers), MergeR(merging received files), Reduce (reduce function), Write (writing result to the HDFS), Cleanup. Due to the fact that the job of MR program carried out in parallel or in waves, it is possible to calculate the approximate total cost of the job through the cost of one task (one mapper and one reducer). The $Cost_{job}$ take into account the parallel threads of execution and compute the total cost of MR job, where c_m and c_r are costs of one task mapper or reducer respectively, MaxMN and MaxRN are maximum map tasks or reduce task per node.

$$Cost_{job} = \frac{|map| * c_m}{|nodes| * MaxMN} + \frac{|red| * c_r}{|nodes| * MaxRN} + c_{tr}$$

This formula is bad for the skew data, when one task is time consuming.

$$c_m = \begin{cases} c_{read} + c_{Map} + c_{Buffer} + c_{merge}, |red| > 0 \\ c_{read} + c_{Map} + c_{Write}, otherwise \end{cases},$$

$$c_r = c_{shuffle} + c_{mergeR} + c_{reduce} + c_{Write}.$$

C_{Map} and c_{reduce} are the cost of user-define functions, so for each join algorithm it is calculated by the own formula. Another cost values from (c_{read}, c_{Buffer}, c_{Write}, c_{merge}, c_{mergeR}, $c_{shuffle}$, c_t) are common for join algorithms. Consider these costs in more detail as [15, 24]. Stages of reading input data from HDFS and writing into HDFS are calculated by:

$$c_{read} = s(split) * r_h, c_{Write} = s(out) * w_h \ ,$$

where split is input split for mapper task, out is the output data of job. The buffering phase is more complicated; during this stage three processes take place: partitioning, sorting and spilling to disk.

$$c_{Buffer} = s(split) * rw_l + p(outm) * (pC + sC + $$
$$+ cC + sortC * \log_2\left(\frac{p(buf)}{|red|}\right)$$

Where outm is output from map functions, buf is buffer for this stage. The buffer is divided into two parts, there are serialization buffer (SB), that contains key-value pairs and an accounting buffer (AB) that contains the metadata. So, the number of pairs in buffer is:

$$p(buf) = \min\{p(SB), p(AB)\}$$

$$p(SB) = \left\lfloor \frac{sortMB * 2^{20} * (1 - sortRP) * sortSP}{wid} \right\rfloor$$

$$p(AB) = \left\lfloor \frac{sortMB * 2^{20} * sortRP * sortSP}{16} \right\rfloor$$

The number of spilled files (N) from this stage is:

$$N = \left\lceil \frac{p(out)}{p(buf)} \right\rceil$$

Then all spilled files must be merged with such features:

- the number of spill files are merged at once is F,
- assume that the following $N \le F^2$,
- at first pass it is merged so spill files that remain files is multiplies F
- at final merge if needed the combiner will be used.

The number of spill files equal to sum of spill files at first pass (S1P), at intermediate pass (SIP) and at final pass (SFP):

$$S1 = \begin{cases} N, N \le F \\ F, (N-1) \bmod (F-1) = 0, \\ (N-1) \bmod (F-1) + 1 \end{cases}$$

$$SIP = \begin{cases} 0, N \le F \\ S1 + \left\lfloor \frac{N - S1}{F} \right\rfloor * F, N \le F^2 \end{cases}$$

$$SFP = \begin{cases} N, N \le F \\ 1 + \left\lfloor \frac{N - S1}{F} \right\rfloor + N - SIP, N \le F^2 \end{cases}$$

$$c_{merge} = p(buf) * wid * rw_l * (2 * SIP + N + N * cC)$$
$$+ p(buf) * mC * (SIP + N)$$

After that stage map output transferred to the reducers (this cost includes the cost for all reducers).

$$c_{tr} = s(outm) * |map| * \frac{|nodes - 1|}{|nodes|} * tC$$

The data from mappers are transferred by segments to reducers. Without considering the data skew, it is assumed that the sizes of segments are the same.

$$s(seg) = \frac{s(outm)}{|red|}$$

When segment arrive to the reducer it is placed in shuffle buffer or if size of segment is greater than 25% of buffer size then it is spilled into disk without in-memory buffer. The buffer size is determined by the configuration parameters as:

$s(buf) = shuBP * memT$. If buffer reaches size threshold (s(thr)) or the number of segments is greater than memMT, then segments are merged, sort and spill into disk. $s(thr) = s(buf) * shuMP$. The number of segments (|segF|) in shuffle file and the number of such files (|shF|) are:

$$|segF| = \begin{cases} 1, s(seg) \ge 0{,}25 * s(buf) \\ \left\lfloor \frac{s(thr)}{s(seg)} \right\rfloor, s(seg) < 0{,}25 * s(buf) \\ memMT, |segF| \ge memMT \end{cases}$$

$$|shF| = \left\lfloor \frac{|map|}{|segF|} \right\rfloor$$

If the number of shuffle files is greater than (2*F-1) then all files are merged into one. So, all segments may be divided on three states: in-memory buffer (segMB), shuffle unmerged files (segUF) and shuffle merged files (segMF).

$$|segMB| = |map| \bmod |segF|$$

$$|segMF| = \begin{cases} 0, |shF| < 2*F-1 \\ \left\lfloor \dfrac{|shF| - 2*F+1}{F} \right\rfloor + 1 \end{cases}$$

$$|segUF| = |shF| - F*|segMF|$$

The cost of shuffle stage is:

$$c_{sfuffle} = |segF| * s(seg) * selC * rw_l * (|shF| + \\ + |segMF| * 2) + |segMF| * |segF| * p(seg) * \\ * mC + |map| * p(seg) * (mC + cC) * I$$

$$I = \begin{cases} 1, s(seg) < 0{,}25 * s(buf) \\ 0 \end{cases}$$

Thereafter, segMB,segUF, segMF files must be merged. Some segments from memory (segE) are spilled to disk by redBP constraint.

$$|segE| = \begin{cases} \left\lceil \dfrac{|segMB| * s(seg) - redBP * memT}{s(seg)} \right\rceil \\ 0, |segMB| * s(seg) < redBP * memT \end{cases}$$

If the number of files from disk is less than F then segE files are merged separately.

$$s(m1) = \begin{cases} |segE| * s(seg) \\ 0, |segUF| + |segMF| \geq F \end{cases}$$

After the merging, the number of files from disk is:

$$|segD| = \begin{cases} |segUF| + |segMF| + 1, s(m1) > 0 \\ |segUF| + |segMF| + |segE| \end{cases}$$

Then the process of merging is similar to c_{merge}, where N=|segD|.

$$s(m2) = \frac{SIP}{N} * (s(segUF) + s(segMF) + s(segE))$$

At final it is merged remained files.

$$|segR| = SFP * (|segMB| - |segE|)$$

$$N = |segR|$$

$$s(m3) = \frac{SIP}{N} * |map| * s(seg)$$

The final cost of this phase is:

$$c_{mergeR} = (s(m1) + s(m2) + s(m3)) * \left(rw_l + \frac{mC}{wid} \right)$$

Since the join algorithms are known in advance we can more accurately than the approach in [28] is to estimate the cost of user-defined functions Map and Reduce.

4.3 Cost model for Reduce-Side join

In case of General reducer-side join, MR program consists of one job and cost for combining is equal to 0.

In map function source tag is assign to each pair (consider that input map pair is equal to output map pair):

$$c_{Map}^{GRSJ} = p(outm) * c_t, wid = wid + 0{,}000000953$$

In reduce function pairs with different tags are joined (nested-loop):

$$c_{reduce}^{GRSJ} = \left(\left(\frac{p(inpr)}{2} \right)^2 * selP + p(inpr) \right) * c_t$$

As opposite to General reducer-side join, the cost of Optimized reducer-side join includes the cost of combine function and the cost of reduce function is less then c_{reduce}^{GRSJ}:

$$c_{reduce}^{ORSJ} = \left(\frac{p(inpr)}{2} + \left(\frac{p(inpr)}{2} \right)^2 * selP \right) * c_t\ ,$$

$$c_{Map}^{ORSJ} = c_{Map}^{GRSJ}, wid = wid + 0{,}00000190734$$

In contrast to the previous join, MR program of the Hybrid Hadoop join consist of pre-processing job and join job. The pre-processing job is partition one dataset into |red| parts, and besides these partitions may be got from other MR job or from default MR job. The costs of default map and reduce functions are:

$$c_{Map}^{prep} = c_{reduce}^{prep} = p(in1) * c_t$$

There are two ways to deliver full one dataset to the mapper: read file from HDFS or by using distributed cache. And if distributed cache is used then the necessary files are copied to the slave nodes before the job is started. So, the c_{tr} cost is added. The costs of with and without distributed cache deliver are:

$$c_{cache} = s(in1) * wr_l + p(in1) * c_t$$

$$c_{hdfs} = s(in1) * r_h + p(in1) * c_t$$

The map and reduce functions costs of join job are:

$$c_{Map}^{hyb} = c_{Map}^{prep}(in2),$$

$$c_{reduce}^{hyb} = \begin{cases} c_{cache} + p(in1) * p(in2) * selP * c_t \\ c_{hdfs} + p(in1) * p(in2) * selP * c_t \end{cases}$$

4.4 Cost model for Map-Side join

The join job doesn't have reducer phase.

Map-side partition join consists of pre-processing jobs for two input datasets (or partitions are got from another job) and join job.

The map function of join job is:

$$c_{reduce}^{MSPJ} = c_{reduce}^{hyb}$$

In-Memory Join the small dataset (in1) is broadcast to all reducers.

$$c_{Map}^{IMMJ} = \begin{cases} c_{cache} + p(in2) * \dfrac{p(in1)}{|red|} * selP * c_t \\ c_{hdfs} + p(in2) * \dfrac{p(in1)}{|red|} * selP * c_t \end{cases}$$

In reversed join the datasets are reversed, in2 (the bigger one) is broadcast, in1 is split of smaller dataset and it is loaded in hash table.

$$c_{Map}^{rev} = \begin{cases} p(in1)*c_t + p(in2)*wr_l + p(in1)* \\ *\dfrac{p(in2)}{|red|}*selP*c_t, cache \\ p(in1)*c_t + p(in2)*r_h + p(in1)* \\ *\dfrac{p(in2)}{|red|}*selP*c_t \end{cases}$$

Multi-phase map join cost equal to sum of immj job costs. The number of summands is $\dfrac{s(in1)}{memT}+1$.

4.5 The semi-join cost

The semi-join with selection consists of two jobs: finding unique keys and filter the dataset by unique keys. The cost of map function of finding unique keys is sum of filling hash table and producing the output costs. The input for this job is one dataset.

$c_{Map}^{find} = p(in1)*2*c_t$. The reduce function of that job is run on the one reducer and the same as default reduce function.

The filtering job consists of one map phase, where the file with unique key from previous job is loaded into hash table and then the split of another dataset is probe.

$$c_{Map}^{fil} = \begin{cases} c_{cache} + p(in)*c_t \\ c_{hdfs} + p(in)*c_t \end{cases}$$

The Adaptive semi-join is similar to reduce-side join. The two datasets are read and tagged by label in map function. And at reducer the pairs with different tags are output. The cost is equal to default job. But at the actual join it is needed to add some cost of loading file with unique keys, filling hash table and filtering useless pairs as c_{Map}^{fil}.

In case of semi-join with bloom-filter the program consists of two jobs: creating bloom filter and filtering the dataset. In the map function, bloom filter for split constructed and the output all filter as one pair.

$$c_{Map}^{bloom} = p(in1)*c_t + s(bloom)*lo$$

Where lo is the cost for processing bloom filter. The reducer is one and it is combine all bloom-filter into one.

$$c_{\text{Reduce}}^{bloom} = |map|*s(bloom)*lo$$

At another job the constructed bloom-filter is loaded and the second dataset is probed.

$$c_{Map}^{filb} = s(bloom)*r_h + p(in2)*c_t$$

5 Experiments

5.1 Dataset

Data are the set of tuples, which attributes are separated by a comma. Tuple is split into a pair of a key and a value, where value is the remaining attributes. Generation of synthetic data was done as in [4]. Join keys are distributed randomly.

5.2 Cluster configuration

Cluster consists of three virtual machines, where one of them is master and slave at the same time, the remaining two are the slaves. Host configuration consists of 1 processor, 512 mb of memory for the master, for others nodes have by 512 mb, 5 gb is the disk size. Hadoop 20.203.0 runs on Ubuntu 10.10.

5.3 The General Case

The base idea of this experiment is to compare executions time of different phases of various algorithms. Some parameters are fixed: the number of Map and Reduce tasks is 3, the input size is 10000*100000 and 1000000*1000000 tuples.

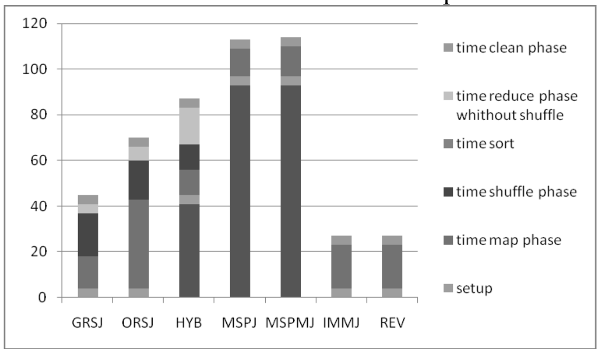

Figure 1: Executions time of different phases of various algorithms. Size 10000*100000.

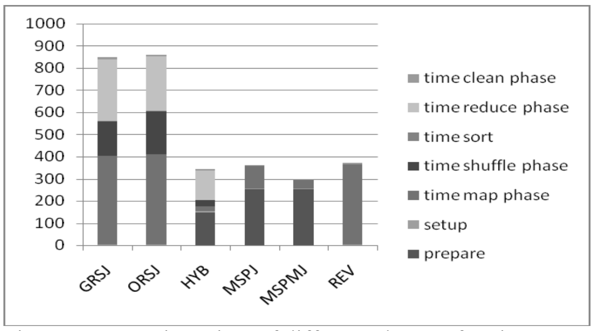

Figure 2: Executions time of different phases of various algorithms. Size 1000000*1000000.

For a small amount of data, Map phase, in which all tuples are tagged, and Shuffle phase, in which data are transferred from one phase to another, are more costly in Reduce-Side joins. It should be noted that GRSJ is better than ORSJ on small data, but it is the same on big data. It is because in first case time does not spend on combining tuples. Possible, on the larger data ORSJ outperform GRSJ when the usefulness of grouping by key will be more significant. Also for algorithms with pre-processing more time are spent on partitioning data. The algorithms in memory (IMMJ and REV) are similar in small data. Two algorithms are not shown in the graph because of their bad times: JDBM-based map join and Multi-phase map join. In large data IMMJ algorithm could not be executed because of memory overflow.

5.4 Semi-Join

The main idea of this experiment is to compare different semi-join algorithms. These parameters are fixed: the number of Map and Reduce tasks is 3, the bitmap size of Bloom-filter is 2500000, the number of hash-functions in Bloom-filter is 173, built-in Jenkins hash algorithm is used in Bloom-filter. Adaptive semi-join (ASGRSJ) does not finish because of memory overflow. The abbreviation of Bloom-filter semi-join for GRSJ is BGRSJ. The abbreviation of semi-join with selection for GRSJ is SGRSJ respectively.

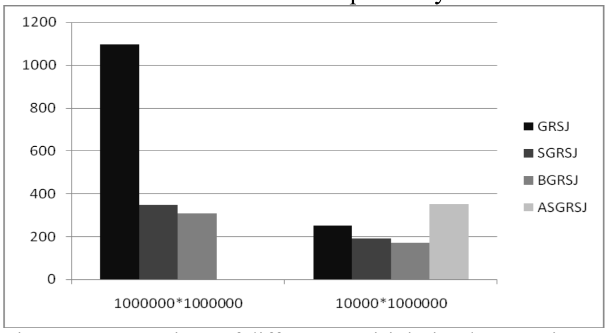

Figure 3: Comparison of different semi-join implementations.

5.5 Distributed cache

In [21] was showed that using of distributed cache is not always good strategy. They suggested that the problem can be a high speed network. This experiment was carried out for Reversed Map-Side join, because for which a distributed cache can be important. Replication was varied as 1, 2, 3 and size of data is fixed – 1000000*1000000 tuples. When data is small, the difference is not always visible. In large data algorithms with distributed cache outperform approach of reading from a globally distributed system.

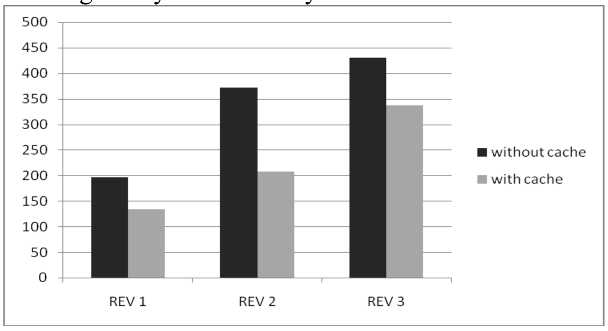

Figure 4: Performance of Reversed Map-Side join with and without using distributed cache.

5.6 Skew data

It is known that many of the presented algorithms are sensitive to the data skew. In this experiment take part such algorithms as Reduce-side join with Simple Range-based Partitioner for GRSJ (GRSJRange) and Virtual Processor Partitionerfor GRSJ (GRSJVirtual), and also for comparing in memory join: IMMJ, REV because of resistant to the skew. Fixed parameters are used: size of two dataset is 2000000, one of the data set has skew 500000 of 5, and another has 10 or 1 of 5. In case with IMMJ was memory overflow.

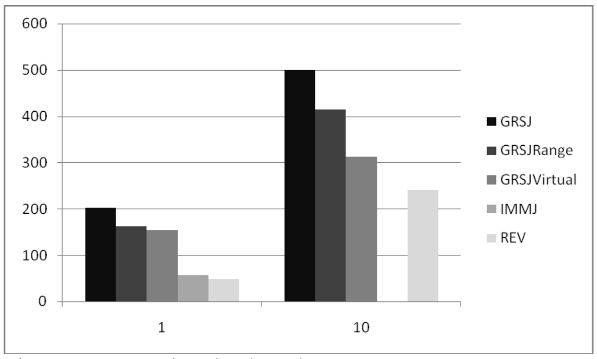

Figure 5: Processing the data skew.

Although these experiments do not completely cover the tuneable set of Hadoop parameters, they are shown the advantages and disadvantages of the proposed algorithms. The main problems of these algorithms are time spent on pre-processing, transferring data, the data skew, and memory overflow.

Each of the optimization techniques introduces additional cost to the implementation of the join, so the algorithm based on the tuneable settings and specific data should be carefully chosen. Also important are the parameters of the network bandwidth when distributed cache are used or not used and a hardware specification of nodes because of it is importance when speculative executions are on. Speculative execution reduces negative effects of non-uniform performance of physical nodes.

Based on the collected statistics such as data size, how many keys will be taking part in the join, these statistics may be collected as well as the construction of a range partitioner, the query planner can choose an efficient variant of the join. For example, in [5] was proposed what-if analyses and cost-based optimization.

6 Future work

The algorithms discussed in this paper, only two sets are joined. It is interesting to extend from binary operation to multi argument joins. Among the proposed algorithms, there is no effective universal solution. Therefore, it is necessary to evaluate the proposed cost models for join algorithms. And for this problem it is need to use real cluster with more than three nodes in it and more powerful to process bigger data, due to the fact that the execution time on the virtual machine may be different from the real cluster in reading/writing, transferring data over the network and so on.

Also the idea of processing the data skew in MapReduce applications from [19] can be applied to the join algorithms. Another direction to future work is to extend algorithm to support a theta-join and outer join. An interesting area for future work is to develop, implement and evaluate algorithms or extended algebraic operations suitable for complex similarity queries in an open distributed heterogeneous environment. The reasons to evaluate complex structured queries are: a need to combine search criteria for different types of information; a query refinement e.g. based on user profile or feedback; advanced users may need query structuring. The execution model and algebraic operation to be implemented are outlined in [31]. The main goal is to solve the problems presented in [8] as a problem.

In addition, one of the issues is efficient physical representation of data. Binary formats are known to outperform the text both in speed reading and partitioning key / value pairs, and the transmission of compressed data over the network. Along with the binary data format, column storage has already been proposed for paradigm MapReduce. It is interesting to find the best representation for specific data.

7 Conclusion

In this work we describe the state of the art in the area of massive parallel processing, presented our comparative study of these algorithms, cost models and our outline directions of future work.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD ’08, pages 967–980, New York, NY, USA, 2008. ACM.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proc. VLDB Endow., 2:922–933, August 2009.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Proceedings of the 13th International Conference on Extending Database Technology, EDBT ’10, pages 99–110, New York, NY, USA, 2010. ACM.

[4] Fariha Atta. Implementation and analysis of join algorithms to handle skew for the hadoop mapreduce framework. Master’s thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2010.

[5] Shivnath Babu. Towards automatic optimization of mapreduce programs. In Proceedings of the 1st ACM symposium on Cloud computing, SoCC ’10, pages 137–142, New York, NY, USA, 2010. ACM.

[6] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 international conference on Management of data, SIGMOD ’10, pages 975–986, New York, NY, USA, 2010. ACM.

[7] A Chatzistergiou. Designing a parallel query engine over map/reduce. Master’s thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2010.

[8] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating db and ir technologies: What is the sound of one hand clapping? In CIDR, pages 1–12, 2005.

[9] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. Proc. VLDB Endow., 2:1481–1492, August 2009.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. Commun. ACM, 53:72–77, January 2010.

[11] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation. USENIX Association, 2004.

[12] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In EDBT 2012, march 2012.

[13] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. Proc. VLDB Endow., 4:419–429, April 2011.

[14] Alan F Gates. Programming Pig. O’Reilly Media, 2011.

[15] Herodotos Herodotou. Hadoop performance models. CoRR, abs/1106.0940, 2011.

[16] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. PVLDB, 4(11):1111– 1122, 2011.

[17] Eaman Jahani, Michael J. Cafarella, and Christopher R´e. Automatic optimization for mapreduce programs. Proc. VLDB Endow., 4:385–396, mar 2011.

[18] Dawei Jiang, Anthony K. H. Tung, and Gang Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. IEEE Transactions on Knowledge and Data Engineering, 23:1299– 1311, 2011.

[19] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. Moskow, Russia, june 2011. In the 5th Open Cirrus Summit.

[20] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In Proceedings of the 2011 international conference on Management of data, SIGMOD ’11, pages 961–972, New York, NY, USA, 2011. ACM.

[21] Gang Luo and Liang Dong. Adaptive join plan generation in hadoop. Technical report, Duke University, 2010.

[22] Christine Morin and Gilles Muller, editors. European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010. ACM, 2010.

[23] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In Proceedings of the 2011 international conference on Management of data, SIGMOD ’11, pages 949–960, New York, NY, USA, 2011. ACM.

[24] Konstantina Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master’s thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2009.

[25] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[26] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD Rec., 18:110–121, June 1989.

[27] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In Proceedings of the 2010 international conference on Management of data, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[28] Vertica Systems, Inc. Managing Big Data with Hadoop & Vertica, 2009.

[29] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In MASCOTS, pages 1–11. IEEE, 2009.

[30] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[31] Anna Yarygina, Boris Novikov, and Natalia Vassilieva. Processing complex similarity queries: A systematic approach. In Maria Bielikova, Johann Eder, and A Min Tjoa, editors, ABDIS 2011 Research Communications: Proceedings II of the 5th East-European Conference on Advances in Databases and Information Systems 20 – 23 September 2011, Vienna, pages 212–221. Austrian Computer Society, September 2011.

[32] Minqi Zhou, Rong Zhang, Dadan Zeng, Weining Qian, and Aoying Zhou. Join optimization in the mapreduce environment for column-wise data store. In Proceedings of the 2010 Sixth International Conference on Semantics, Knowledge and Grids, SKG '10, pages 97–104, Washington, DC.