Block-based Load Balancing for Entity Resolution with MapReduce

Lars Kolb, Andreas Thor, Erhard Rahm
Database Group
University of Leipzig, Germany
{kolb,thor,rahm}@informatik.uni-leipzig.de

ABSTRACT

The effectiveness and scalability of MapReduce-based implementations of complex data-intensive tasks depend on an even redistribution of data between map and reduce tasks. In the presence of skewed data, sophisticated redistribution approaches thus become necessary to achieve load balancing among all reduce tasks to be executed in parallel. For the complex problem of entity resolution with blocking, we propose BlockSplit, a load balancing approach that supports blocking techniques to reduce the search space of entity resolution. The evaluation on a real cloud infrastructure shows the value and effectiveness of the proposed approach.

1. INTRODUCTION

Cloud computing has become a popular paradigm for efficiently processing computational- and data-intensive tasks. Such tasks can be executed on demand on powerful distributed hardware and service infrastructures. The MapReduce (MR) model [2] supports the transparent parallel execution of complex tasks on cloud infrastructures. However, the (cost-) effectiveness and scalability of MR programs depend on effective load balancing approaches to evenly utilize available nodes. This is particularly challenging for data-intensive tasks where skewed data redistribution may cause node-specific bottlenecks and load imbalances.

We propose and evaluate an effective load balancing approach to data skew handling for MR-based entity resolution (ER). Note that MR's inherent vulnerability to load imbalances due to data skew is relevant for all kind of pairwise similarity computation, e.g., document similarity computation [4] and set-similarity joins [9]. Such applications can therefore also benefit from our load balancing approach though we study MR-based load balancing in the context of ER only. ER is the task of identifying entities referring to the same real-world object [7]. It is a pervasive problem and of critical importance for data quality and data integration, e.g., to match product offers for price comparison portals. ER techniques usually compare pairs of entities by evaluating multiple similarity measures. Naïve approaches examine the Cartesian product of n input entities. However, the resulting complexity of $O(n^2)$ is inefficient for large datasets. The common approach to improve efficiency is to adopt so-called blocking techniques [1]. They utilize a blocking key based on the values of one or several entity attributes to partition the input data into multiple partitions (blocks) and restrict the subsequent matching to entities of the same block. For example, it is sufficient to compare entities of the same manufacturer when matching product offers.

Even with blocking, ER remains a costly process that can take up to days for large datasets [8]. The MR model is well suited to execute blocking-based ER in parallel. Several map tasks can read the input entities in parallel and redistribute them among several reduce tasks based on the blocking key. This guarantees that all entities of the same block are assigned to the same reduce task so that different blocks can be matched in parallel by multiple reduce tasks. However, such a basic MR implementation is susceptible to severe load imbalances due to skewed block sizes since the match work of an entire block is assigned to a single reduce task. As a consequence, large blocks prevent the utilization of more than a few nodes. The absence of skew handling mechanisms can therefore tremendously deteriorate runtime efficiency and scalability of MR programs. Furthermore, idle nodes may produce unnecessary costs because public cloud infrastructures usually charge per utilized machine hours.

We therefore propose BlockSplit, a general load balancing approach that addresses the mentioned skew problems. It takes the size of blocks into account and assigns entire blocks to reduce tasks if this does not violate load balancing constraints. Larger blocks are split into smaller chunks based on the input partitions to enable their parallel matching within multiple reduce tasks. The evaluation uses real-world data and demonstrates the importance of skew handling for MR-based ER and the efficiency of BlockSplit.

2. MAPREDUCE AND ENTITY RESOLUTION

MapReduce (MR) is a programming model designed for parallel data-intensive computing in cluster environments with up to thousands of nodes [2]. A computation is expressed with two user defined functions, map and reduce, that are executed in parallel on disjoint partitions of the input data. The map function transforms input entities to key-value pairs whereas reduce is called for each key that occurs as map output. Within the reduce function, one can access the list of all corresponding values. A MR cluster consists

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.
Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...$10.00.

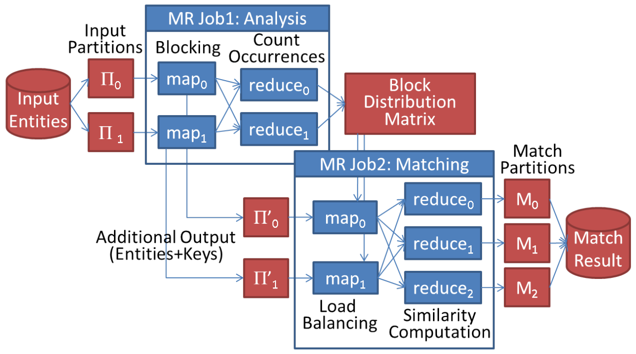

Figure 1: Schematic overview of the MR-based matching process with load balancing.

of a set of nodes that run a fixed number of map and reduce processes. For each MR job execution, the number of map tasks (m) and reduce tasks (r) is specified. Each process can execute only one task at a time. After a task has finished, another task is automatically assigned to the released process using a framework-specific scheduling mechanism.

Besides map and reduce, a MR dataflow relies on three further functions. First, the function part partitions the map output and thereby distributes it to the available reduce tasks. All keys are sorted with the help of a comparison function comp. Finally, each reduce task employs a grouping function group to determine the data chunks for each reduce function call. Each of these functions operates only on (parts of) the key of key-value pairs and does not take the values into account. The use of extended (composite) keys and an appropriate choice of part, comp, and group supports sophisticated partitioning and grouping behavior and will be utilized in our load balancing approach.

As discussed in the introduction, parallel ER using blocking can be easily implemented with MR. The map function can be used to determine for every input entity its blocking key and to output a key-value pair (blocking_key, entity). The default hash partitioning strategy would use the blocking key to distribute key-value pairs among reduce tasks so that all entities sharing the same blocking key are assigned to the same reduce task. Finally, the reduce function is called for each block and computes the matching entity pairs within its block. We call this straightforward approach Basic. However, the Basic strategy is vulnerable to data skew due to blocks of varying size. Therefore, the execution time may be dominated by a single or a few reduce tasks.

3. BLOCK-BASED LOAD BALANCING

We describe our load balancing approach BlockSplit for ER for one data source R. The input is a set of entities and the output are the entity pairs that are considered to be the same. We perform ER processing within two MR jobs as illustrated in Figure 1. Both jobs are based on the same number of map tasks and the same partitioning of the input data. The first job calculates a so-called block distribution matrix (BDM) that specifies the number of entities per block separated by input partitions. The matrix is used by the map phase of the second MR job to tailor entity redistribution for parallel matching of blocks of different size.

BlockSplit generates one or several so-called match tasks per block and distributes match tasks among reduce tasks. Furthermore, it uses the following two ideas:

- Small blocks are processed within a single match task similar to Basic. Large blocks are split according to the m input partitions into m sub-blocks. All entities within the same sub-block are compared with each other. Furthermore, pairs of sub-blocks are processed to evaluate their Cartesian product. This ensures that all comparisons of the original block will be computed.
- BlockSplit determines the number of comparisons per match task and assigns match tasks in descending size among reduce tasks. This implements a greedy load balancing heuristic ensuring that the largest match tasks are processed first and makes it unlikely that two large match tasks are processed by the same reduce task.

In the following, we describe the computation of the BDM and our BlockSplit strategy. Further information incl. pseudocode can be found in [5].

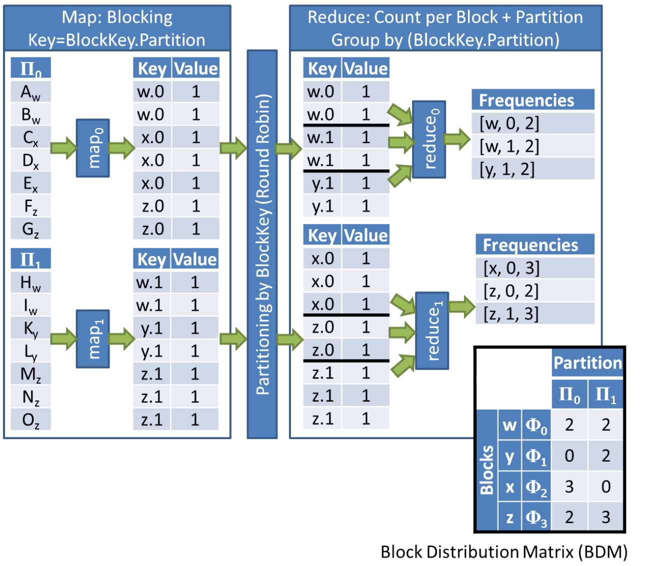

Figure 2: Example dataflow for computation of the block distribution matrix (MR Job1 of Figure 1).

3.1 Block Distribution Matrix

The block distribution matrix (BDM) is a $b \times m$ matrix that specifies the number of entities of b blocks across m input partitions. The BDM computation using MR is straightforward as illustrated by Figure 2 for an example dataset. The 14 entities (A-O) are divided into two input partitions Π_0 and Π_1 and each entity has a blocking key (w-z) that is denoted as index. For example, the map output key of M is $z.1$ because M's blocking key equals z and M appears in the second partition (partition index=1). This key is assigned to the last reduce task that outputs $[z, 1, 3]$ because there are 3 entities in the second partition for blocking key z. The combined reduce outputs correspond to the BDM. To assign block keys to rows of the BDM, we use the order of the blocks from the reduce output, i.e., we assign the first block (key w) to block index position 0, etc.

The block sizes in the example vary between 2 and 5 entities. The match work to compare all entities per block with each other thus ranges from 1 to 10 comparisons; the largest

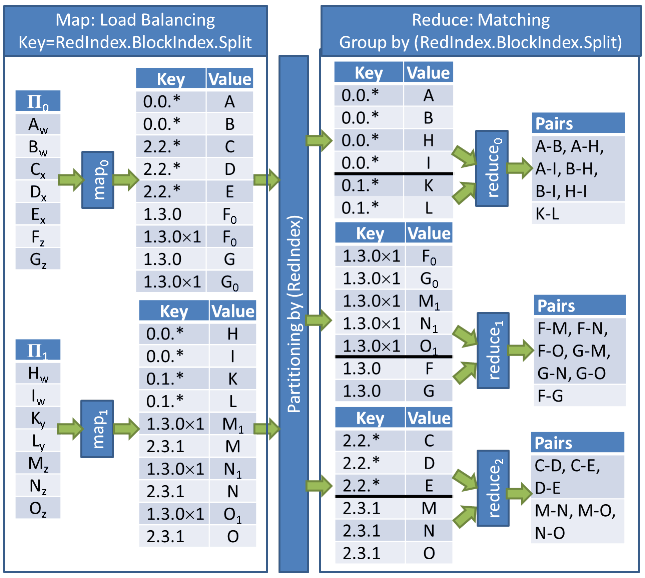

Figure 3: Example dataflow for the load balancing strategy BlockSplit (MR Job2 of Figure 1).

block with key z entails 50% (10 of 20) of all comparisons although it contains only 35% (5 out 14) of all entities.

As illustrated in Figure 1, map produces an additional output Π'_i per input partition that contains the original entities annotated with their blocking keys. This output is not shown in Figure 2 to save space but used as input in the second MR job (see Figure 3).

3.2 BlockSplit

The realization of BlockSplit makes use of the BDM as well as of composite map output keys. Map generates a carefully constructed composite key that (together with associated partition and group functions) allows a balanced load distribution. The composite key thereby combines information about the target reduce task(s), the block of the entity, and the entity itself. The map function may generate multiple keys per entity if this entity is supposed to be processed by multiple reduce tasks. The reduce phase performs the actual ER and computes match similarities between entities of the same block. Since the reduce phase consumes the vast majority of the overall runtime (~95% in our experiments), our load balancing strategy solely focusses on data redistribution for reduce tasks. Other MR-specific performance factors as data locality are therefore not considered.

Map outputs key-value pairs with key=(reduce_index $\odot$ block_index $\odot$ split) and value=(entity). The reduce task index has a value between 0 and $r-1$ and is used by part to realize the desired assignment to reduce tasks. Grouping is done on the entire key and – since the block index is part of the key – ensures that each reduce function only receives entities of the same block. The split value indicates what match task has to be performed by the reduce function, i.e., whether a complete block or sub-blocks need to be processed.

During the initialization, each of the m map tasks reads the BDM and computes the number of comparison per block and the total number of comparisons P over all b blocks Φ_k: $P = \frac{1}{2} \cdot \Sigma_{k=0}^{b-1} |\Phi_k| \cdot (|\Phi_k| - 1)$. For each block Φ_k, it also checks if the number of comparisons is above the average reduce task workload, i.e., if

$$\frac{1}{2} \cdot |\Phi_k| \cdot (|\Phi_k| - 1) > P/r.$$

If the block Φ_k is *not* above the average workload it can be processed within a single match task (denoted as $k.*$ in the block_index and split components of the map output key). Otherwise, it is split into m sub-blocks based on the m input partitions leading to the following $\frac{1}{2} \cdot m \cdot (m-1) + m$ match tasks:

- m match tasks, denoted with key components $k.i$, for the individual processing of the i^{th} sub-block for $i \in [0, m-1]$
- $\frac{1}{2} \cdot m \cdot (m-1)$ match tasks, denoted with key components $k.i{\times}j$ with $i, j \in [0, m-1]$ and $i < j$, for the computation of the Cartesian product of sub-blocks i and j

Note that the BDM holds the number of entities per (block, partition) pair and map can therefore determine which input partitions contain *no* entities of Φ_k, resulting in less match tasks. However, in favor of readability we assume that all m input partitions contain at least one entity. To determine the reduce task for each match task, all match tasks are first sorted in descending order of their number of comparisons. Match tasks are then assigned to reduce tasks in this order so that the current match task is assigned to the reduce task with the lowest number of already assigned pairs. In the following, we denote the reduce task index for match task $k.x$ with *R(k.x)*.

After the described initialization phase, the map function is called for each input entity. If the entity belongs to a block Φ_k that has *not* to be split, map outputs one key-value pair with composite key=*R(k.*).k.**. Otherwise, map outputs m key-value pairs for the entity. The key *R(k.i).k.i* represents the individual sub-block i of block Φ_k and the remaining $m-1$ keys $R(k.i).k.i \times j$ (for $j \in [0, m-1]$ and $j \neq i$) represent all combinations with the other $m-1$ sub-blocks. This indicates that entities of split blocks are replicated m times to support load balancing. The map function emits the entity as value of the key-value pair; for split blocks we annotate entities with the partition index for use in the reduce phase.

In our running example, only block Φ_3 (blocking key z) is subject to splitting into m=2 sub-blocks. The BDM (see Figure 2) indicates for block Φ_3 that Π_0 and Π_1 contain two and three entities, respectively. The resulting sub-blocks $\Phi_{3.0}$ and $\Phi_{3.1}$ lead to the three match tasks *3.0*, *3.0×1*, and *3.1* that account for 1, 6, and 3 comparisons, respectively. The resulting ordering of match tasks by size (*0.**, *3.0×1*, *2.**, *3.1*, *1.**, and *3.0*) leads for three reduce tasks to the distribution shown in Figure 3. The replication of the five entities for the split block leads to 19 key-value pairs for the 14 input entities. Each reduce task has to process between six and seven comparisons indicating a good load balancing for the example.

4. EVALUATION

In the following we evaluate our BlockSplit strategy regarding the degree of data skew and the number of available nodes (n). We ran our experiments with real-world datasets on the Amazon EC2 cloud infrastructure using Hadoop.

We first evaluate the robustness of our load balancing strategies against data skew for a dataset containing about 114,000 product descriptions. To this end, we control the degree of data skew by generating block distributions that

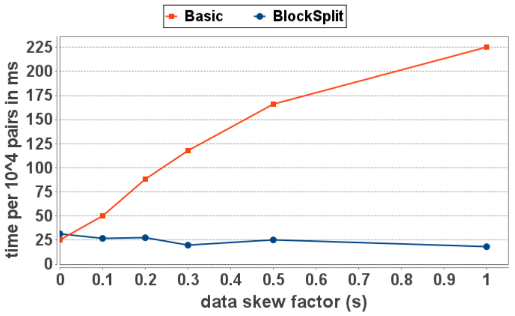

Figure 4: Execution times for different data skews using 114,000 entities.

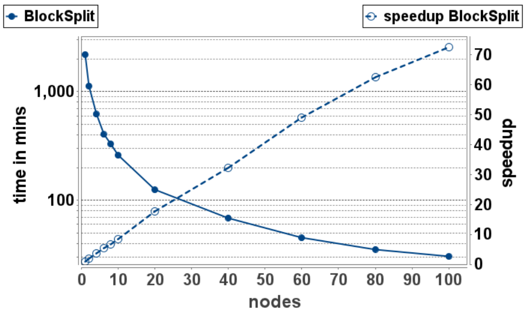

Figure 5: Execution times and speedup of the BlockSplit strategy.

follow an exponential distribution. Given a fixed number of blocks b=100, the number of entities in the k^{th} block is proportional to $e^{-s \cdot k}$. The skew factor $s \geq 0$ thereby describes the degree of data skew. Note that the data skew, i.e., the distribution of entities over all blocks, determines the overall number of entity pairs. We are therefore interested in the average execution time *per entity pair* when comparing load balancing strategies for different data skews.

Figure 4 shows the average execution time per 10^4 pairs for different data skews (n=10, m=20, r=100). The Basic strategy explained in Section 2 is not robust against data skew because a higher data skew increases the number of pairs of the largest block. For example, for s=1 Basic needs 225ms per 10^4 comparisons which is more than 12 times slower than BlockSplit. However, the Basic strategy is slightly faster for a uniform block distribution (s=0) because it does not suffer from the additional BDM computation and load balancing overhead. The BDM influence becomes insignificant for higher data skews because the data skew does not affect the time for BDM computation but the number of pairs. This is why the execution time per pair is reduced for increasing s. In general, BlockSplit is stable across all data skews.

To analyze the scalability of BlockSplit, we use a larger dataset of approx. 1.4 million publication records. The blocking is formed by the first three letters of the publication title. We vary the number of nodes from 1 up to 100. For n nodes, the number of map tasks is set to $m = 2 \cdot n$ and the number of reduce tasks is set to $r = 10 \cdot n$, i.e., adding new nodes leads to additional map and reduce tasks.

The resulting execution times and speedup values are shown in Figure 5. In contrast to Basic, BlockSplit shows its ability to evenly distribute the workload across available nodes. It scales almost linearly up to 40 nodes. For larger n the speedup increase slightly alleviates due to the decreasing workload (number of entity comparisons) per reduce task. At the same time, the relative fraction of the MR overhead for task initialization and task shutdown is increasing.

5. RELATED WORK

Load balancing and skew handling are well-known data management problems and MR has been criticized for having overlooked the skew issue [3]. MR's inherent vulnerability to data skew is relevant for all kind of pairwise similarity computation, e.g., pairwise document similarity [4] or set-similarity joins [9]. MR has already been employed for ER (e.g., [10]) but we are only aware of one load balancing mechanism. [6] studies load balancing for Sorted Neighborhood that is by design less vulnerable to skewed data.

6. SUMMARY AND OUTLOOK

We proposed a load balancing approach, BlockSplit, for parallelizing blocking-based entity resolution using the widely available MapReduce framework. The approach is capable to deal with skewed blocking key distributions and effectively distributes the workload among all reduce tasks by splitting large blocks. Our evaluation in a real cloud environment demonstrated that it is robust against data skew and scales with the number of available nodes.

In future work we will extend our approach to multi-pass blocking that assigns multiple blocks per entity. We will further investigate how our load balancing approach can be adapted for MapReduce-based implementations of other data-intensive tasks, such as join processing or data mining.

7. REFERENCES

[1] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Workshop Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[3] D. J. DeWitt and M. Stonebraker. MapReduce: A major step backwards, 2008.

[4] T. Elsayed, J. Lin, and D. W. Oard. Pairwise Document Similarity in Large Collections with MapReduce. In *ACL*, 2008.

[5] L. Kolb, A. Thor, and E. Rahm. Load Balancing for MapReduce-based Entity Resolution. *ArXiv*, 2011.

[6] L. Kolb, A. Thor, and E. Rahm. Multi-pass Sorted Neighborhood Blocking with MapReduce. *CSRD*, 2011.

[7] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2), 2010.

[8] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1), 2010.

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

[10] C. Wang et al. MapDupReducer: Detecting near duplicates over massive datasets. In *SIGMOD*, 2010.