Multi-pass Sorted Neighborhood Blocking with MapReduce

Lars Kolb
University of Leipzig
Database Group

Andreas Thor
University of Leipzig
Database Group

Erhard Rahm
University of Leipzig
Database Group & WDI-Lab

{kolb, thor, rahm}@informatik.uni-leipzig.de

Preprint, accepted for publication in "Computer Science - Research and Development"

May 16, 2011

Abstract

Cloud infrastructures enable the efficient parallel execution of data-intensive tasks such as entity resolution on large datasets. We investigate challenges and possible solutions of using the MapReduce programming model for parallel entity resolution using Sorting Neighborhood blocking (SN). We propose and evaluate two efficient MapReduce-based implementations for single- and multi-pass SN that either use multiple MapReduce jobs or apply a tailored data replication. We also propose an automatic data partitioning approach for multi-pass SN to achieve load balancing. Our evaluation based on real-world datasets shows the high efficiency and effectiveness of the proposed approaches.

1 Introduction

Cloud computing has become a popular paradigm for efficiently processing data and computationally intensive application tasks [1]. Many cloud-based implementations utilize the MapReduce programming model for parallel processing on cloud infrastructures with up to thousands of nodes [9]. The broad availability of MapReduce distributions such as Hadoop makes it attractive to investigate its use for the efficient parallelization of data-intensive tasks.

Entity resolution (also known as object matching, deduplication, or record linkage) is such a data-intensive and performance critical task that can likely benefit from cloud computing. Given one or more data sources, entity resolution is applied to determine all entities referring to the same real world object [14, 22]. It is of critical importance for data quality and data integration, e.g., to find duplicate customers in enterprise databases or to match product offers for price comparison portals.

Many approaches for entity resolution have been proposed [2, 12, 18, 20] and included in matching frameworks [4, 6]. The standard (naïve) approach to find matches in n input entities is to apply matching techniques on the Cartesian product of input entities. However, the resulting quadratic complexity of $O(n^2)$ results in infeasible execution times for large datasets [19]. So-called blocking techniques [3] thus become necessary to reduce the number of entity comparisons whilst maintaining match quality. This is achieved by semantically partitioning the input data into blocks of similar records and restricting entity resolution to entities of the same block. Sorted neighborhood (SN) is one of the most popular blocking approaches [14]. It sorts all entities using an appropriate blocking key and only compares entities within a predefined distance window w. The SN approach thus reduces the complexity to $O(n \cdot w)$ for the actual matching. The multi-pass variant of SN utilizes several blocking keys for improved effectiveness [14].

In this study we investigate the use of MapReduce for the parallel execution of single- and multi-pass SN blocking and entity resolution. By combining the use of blocking and parallel processing we aim at a highly efficient entity resolution implementation for very large datasets. The proposed approaches consider specific partitioning requirements of the MapReduce model and implement a correct sliding window evaluation of entities. Our contributions can be summarized as follows:

- We demonstrate how the MapReduce model can be applied for the parallel execution of a general entity resolution workflow consisting of a blocking and matching strategy.
- We identify the major challenges and propose two approaches for realizing SN Blocking on MapReduce. The approaches (called JobSN and RepSN) either use multiple MapReduce jobs or apply a tailored data replication during data redistribution. We also describe an extension for multi-pass SN that allows for a simultaneous employment of multiple blocking functions.
- We address the data skew problem with an automatic data partitioning approach that can be combined with

both JobSN and RepSN. It ensures load balancing across all available nodes and supports multiple SN passes with possibly different window sizes.

- We evaluate our approaches and demonstrate their efficiency in comparison to sequential SN. The evaluation also considers the influence of the window size and data skew as well as the match quality in terms of F-measure.

This paper is an extended version of [17] and introduces support for multi-pass SN as well as automatic data partitioning. The evaluation is expanded by respective experiments.

The rest of the paper is organized as follows. In the next section we introduce the MapReduce programming paradigm. Section 3 illustrates the general realization of entity resolution using MapReduce. In Section 4, we describe our approaches to realize single- and multi-pass SN blocking based on MapReduce. Section 5 explains our method for automatic data partitioning and load balancing. Section 6 describes the performed experiments and evaluation. Related work is discussed in Section 7 before we conclude.

2 MapReduce

MapReduce is a programming model introduced by Google in 2004 [8]. It supports parallel data-intensive computing in cluster environments with up to thousands of nodes. A MapReduce program relies on data partitioning and redistribution. Entities are represented by $(key, value)$ pairs. A computation is expressed with two user defined functions:

$$map : (key_{in}, value_{in}) \rightarrow list(key_{tmp}, value_{tmp})$$
$$reduce : (key_{tmp}, list(value_{tmp})) \rightarrow list(key_{out}, value_{out})$$

These functions contain sequential code and are executed in parallel across many nodes utilizing present data parallelism. Map tasks scan disjoint input partitions in parallel and transform each entity in a $(key, value)$-representation before the *map* function is executed. The output of a *map* function is sorted by key and repartitioned by applying a partitioning function on the key. A partition may contain different keys but all values with the same key are in the same partition. The partitions are redistributed, i.e., all $(key, value)$ pairs of a partition are sent to exactly one reduce task. Each reduce task employs a grouping function to determine the data chunks for each *reduce* function call.

An exemplary data flow of a MapReduce computation is shown in Figure 1. The MapReduce program counts the number of term occurrences across multiple documents which is a common task in information retrieval. The input data (list of documents) is partitioned and distributed

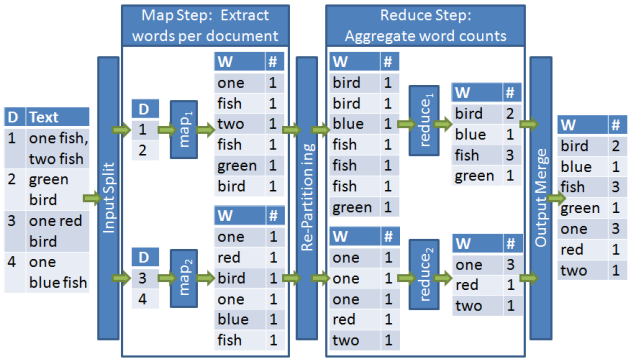

Figure 1: Example of a MapReduce program for counting word occurrences in documents (similar to [21]).

to the two map tasks. In the simple example of Figure 1, two documents are assigned to each of the two map tasks. However, a map tasks usually processes larger partitions in practice. Instances of the map function are applied to each partition of the input data in parallel. In our example, the map function extract all words for all documents and emits a list of $(term, 1)$ pairs. The partitioning assigns every (*key*,*value*) pair to one reduce task according to the key. In the example of Figure 1 a simple range partitioning is applied. All keys (words) starting with a letter from *a* through *m* are assigned to the first reduce task; all other keys are transferred to the second reduce task. The input partitions are sorted for all reduce tasks. The user-defined reduce function then aggregates the word occurrences and outputs the number of occurrences per word. The output partitions of reduce can then easily be merged to a combined result since two partitions do not share any key.

There are several frameworks that implement the MapReduce programming model. Hadoop [13] is the most popular implementation throughout the scientific community. It is free, easy to setup, and well documented. We therefore implemented and evaluated our approaches with Hadoop. A Hadoop cluster consists of a set of nodes that run a number $\hat{m}$ of map and a number $\hat{r}$ of reduce processes. These numbers may change during the execution of a MapReduce program due to node failures or newly added nodes at runtime. More important for our purposes is the specified number of map tasks (m) and reduce tasks (r). Note that the partitioning function relies on the number of reduce tasks since it assigns key-value pairs to the available reduce tasks. Each process can execute only one task at a time. After a task has finished, another task is automatically assigned to the finished process using a Hadoop-specific scheduling mechanism. The number of map (m) and reduce tasks (r) remains unchanged during the execution and is used by our algorithms as an execution parameter. Most MapReduce implementations utilize a distributed file system (DFS) such as the Hadoop distributed file sys-

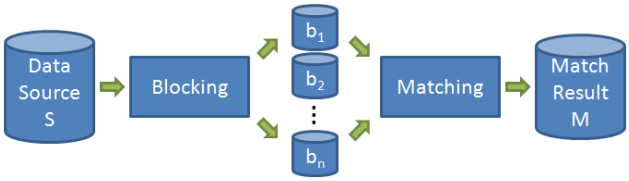

Figure 2: Simplified general entity resolution workflow

tem [5]. The input data is initially stored partitioned, distributed, and replicated across the DFS. Partitions are redistributed across the DFS in the transition from map to reduce. The output of each reduce call is written to the DFS.

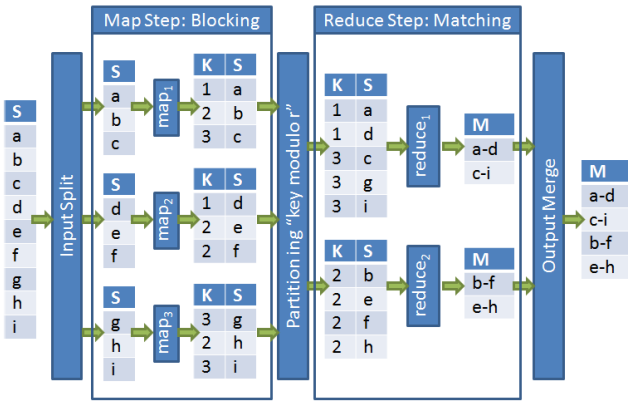

Figure 3: Example of a general entity resolution workflow with MapReduce ($n = 9$ input entities, $m = 3$ map and $r = 2$ reduce tasks)

3 Entity resolution with MapReduce

In this work we consider the problem of entity resolution (deduplication) within one source. The input data source $S = \{e_i\}$ contains a finite set of entities e_i. The task is to identify all pairs of entities $M = \{(e_i, e_k) \mid e_i, e_k \in S\}$ that are regarded as duplicates.

Figure 2 shows a simplified generic entity resolution workflow. The workflow consists of a blocking strategy and a matching strategy. Blocking semantically divides a data source S into possibly overlapping partitions (blocks) b_i, with $S = \bigcup b_i$. The goal is to restrict entity comparison to pairs of entities that reside in the same block. The partitioning into blocks is usually done with the help of blocking keys based on the entities' attribute values. Blocking keys utilize the values of one or several attributes, e.g., product manufacturer (to group together all products sharing the same manufacturer) or the combination of manufacturer and product type. Often, the concatenated prefixes of a few attributes form the blocking key. A possible blocking key for publications could be the combination of the first letters of the authors' last names and the publication year.

The matching strategy identifies pairs of matching entities of the same block. Matching is usually realized by pairwise similarity computation of entities to quantify the degree of similarity. A matching strategy may also employ several matchers and combine their similarity scores. As a last step the matching strategy classifies the entity pairs as match or non-match. Common techniques include the application of similarity thresholds, the incorporation of domain-specific selection rules, or the use of training-based models. Our entity resolution model abstracts from the actual matcher implementation and only requires that the matching strategy returns the list of matching entity pairs.

The realization of the general entity resolution workflow with MapReduce is relatively straightforward by implementing blocking within the map function and by implementing matching within the reduce function. To this end, map first determines the blocking key for each entity. The MapReduce framework groups entities with the same blocking key to blocks and redistributes them. The reduce step then matches the entities within one block. Such a procedure shares similarities with the join computation in parallel database systems [10]. There, the join key (instead of the blocking key) is used for data repartitioning to allow a subsequent parallel join (instead of match) computation. The join (match) results are disjoint by definition and can thus easily merged to obtain the complete result.

Figure 3 illustrates an example for n=9 entities, a-i, of an input data source S using m=3 map and r=2 reduce tasks. First, the input partitioning (split) divides the input source S into m partitions and assigns one partition to each map task. Then, the individual map tasks read their (preferably) local data in parallel and determine a blocking key value K for each of the input entities.[1] For example, entity a has blocking key value 1. Afterwards all entities are dynamically redistributed by a partition function such that all entities with the same blocking key value are sent to the same reduce task (node). In the example of Figure 3, blocking key values 1 and 3 are assigned to the first reduce task whereas key 2 is assigned to the second node. The receivers group the incoming entities locally and identify the duplicates in parallel. For example, the first reduce task identifies the duplicate pairs (a,d) and (c,i). The reduce outputs can finally be merged to achieve the overall match result.

Unfortunately, the sketched MapReduce-based entity resolution workflow has several limitations:

Disjoint data partitioning: MapReduce uses a partition function that determines a single output partition for each map output pair based on its key value. This ap-

[1] Figure 3 omits the map input keys for simplicity.

proach is suitable for many blocking techniques but complicates the realization of blocking approaches with overlapping blocks. For example, the sorted neighborhood approach does not only compare entities sharing the same blocking key.

Load balancing: Blocking may lead to partitions of largely varying size due to skewed key values. Therefore the execution time may be dominated by a single or a few reduce tasks similar to skew effects during parallel join processing [11].

Memory bottlenecks: All entities within the same block are passed to a single reduce call. The reduce task can only process the data row-by-row similar to a forward SQL cursor. It does not have any other options for data access. On the other hand, the matching requires that all entities within the same reduce block are compared with each other. The reduce task must therefore store all entities in main memory (or must make use of other external memory) which can lead to serious memory bottlenecks. The memory bottleneck problem is partly related to the load balancing problem since skewed data may lead to large blocks which tighten the memory problem. Possible solutions have been proposed in [23]. However, memory issues can also occur with a (perfect) uniform key distribution.

In this work we focus on the first and second challenge for the popular and efficient Sorted Neighborhood (SN) blocking method. With respect to the first problem, we propose two MapReduce-based approaches in the next section. In Section 5 we propose a load balancing approach for SN that avoids skew effects. This approach also reduces the risk of memory bottlenecks.

4 Sorted Neighborhood with MapReduce

Sorted neighborhood (SN) [14] is a popular blocking approach that works as follows. A blocking key K is determined for each of n entities. Typically the concatenated prefixes of a few attributes form the blocking key. Afterwards the entities are sorted by this blocking key. A window of a fixed size w is then moved over the sorted records and in each step all entities within the window, i.e., entities within a distance of $w-1$, are compared.

Figure 4 shows a SN example execution for a window size of $w = 3$. The input set consists of the same $n = 9$ entities that have already been employed in the example of Figure 3. The entities (a-i) are first sorted by their blocking keys (1, 2, or 3). The sliding window then starts with the first block (a,d,b) resulting in the three pairs (a,d), (a,b), and (d,b) for later comparisons. The window is then moved by one step to cover the block (d,b,e). This leads

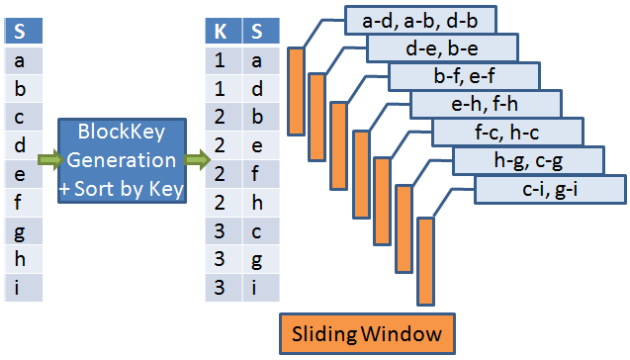

Figure 4: Example execution of sorted neighborhood with window size $w = 3$

to two additional pairs (d,e) and (b,e). This procedure is repeated until the window has reached the final block (c,g,i). Figure 4 lists all pairs generated by the sliding window. In general, the overall number of entity comparisons is $(n - w/2) \cdot (w-1)$.

The SN approach is very popular for entity resolution due to several advantages. First, it reduces the complexity from $O(n^2)$ (matching n input entities without blocking) to $O(n) + O(n \cdot \log n)$ for blocking key determination and sorting and $O(n \cdot w)$ for matching. Thereby matching large datasets becomes feasible and the window size w allows for a dedicated control of the runtime. Second, the SN approach is relatively robust against a suboptimal choice of the blocking key since it is able to compare entities with a different (but similar) blocking key. The SN approach may also be repeatedly executed using different blocking keys. Such a multi-pass strategy diminishes the influence of poor blocking keys (e.g., due to dirty data) whilst still maintaining the linear complexity for the number of possible matches. Finally the linear complexity makes SN more robust against load balancing problems, e.g., if many entities share the same blocking key.

The major difference of SN in comparison to other blocking techniques is that a matcher does not necessarily only compare entities sharing the same blocking key. For example, entities d and b have different blocking keys but need to be compared according to the sorted neighborhood approach (see Figure 4). On the other hand, one of the key concepts of MapReduce is that map input partitions are processed independently. This allows for a flexible parallelization model but makes it challenging to group together entities within a distance of w since a map task has no access to the input partition of other map tasks.

Even if we assume that a map task can determine the relevant entity sets for each entity[2], the general approach as presented in Section 3 is not suitable.

[2]For example, this could be realized by employing a single map task only.

This is due to the fact that the sliding window approach of SN leads to heavily overlapping entity sets for later comparison. In the example of Figure 4, the sliding window produces the blocks $\{a,d,b\}$ and $\{d,b,e\}$ among others. The general MapReduce-based entity resolution approach is, of course, applicable, but would expend unnecessary resources. First of all, almost all entities appear in w blocks and would therefore appear w times in the map output. Finally, the overlapping blocks would cause the generation of duplicate pairs in the reduce step, e.g., (d,b) in the above mentioned example.

We therefore target a more efficient MapReduce-based realization of SN and, thus, adapt the approach described in Section 3. The map function determines the blocking key for each input entity independently. The map output is then distributed to multiple reduce task that implement the sliding window approach for each reduce partition. For example, in the case of two reduce task one may want to send all entities of Figure 4 with blocking key ≤ 2 to the first and the remaining entities to the second reduce task. The analysis of this scenario reveals that we have to solve mainly two challenges to implement a MapReduce-based SN approach. Furthermore, we need to support multiple blocking keys within a multi-pass SN implementation.

Sorted reduce partitions: The SN approach assumes an ordered list of all entities based on their blocking keys. A repartitioning must therefore preserve this order, i.e., the map output has to make sure that all entities assigned to reduce task R_x have a smaller (or equal) blocking key than all entities of reduce task R_{x+1}. This allows each reduce task to apply the sliding window approach on its partition. We will address the sorted data repartitioning by employing a composite key approach that relies on a partition prefix (see Section 4.1).

Boundary entities: The continuous sliding window of SN requires that not only entities within a reduce partition but also across different reduce partitions have to be compared. More precisely, the highest $v < w$ entities of a reduce partition R_x need to be compared with the $w-v$ smallest entities of the succeeding partition R_{x+1}. In the following, we call those entities *boundary entities*. For simplicity we assume that there is no partition that holds less than w entities. Therefore it is sufficient to only compare entities of two succeeding reduce tasks what is surely the common case. We propose two approaches (JobSN and RepSN) that employ multiple MapReduce computation steps and data replication, respectively, to process boundary entities and, thus, to map the entire SN algorithm to a MapReduce computation (Sections 4.2 and 4.3).

Multi-pass SN: SN can be applied for multiple blocking keys and such a multi-pass approach has been shown to significantly improve the effectiveness compared to the use of a single blocking key [14]. We therefore need an efficient MapReduce-based implementation for multi-pass SN. In Section 4.4 we outline such an approach that avoids reading the input data multiple times but uses a single map phase for all blocking keys (passes).

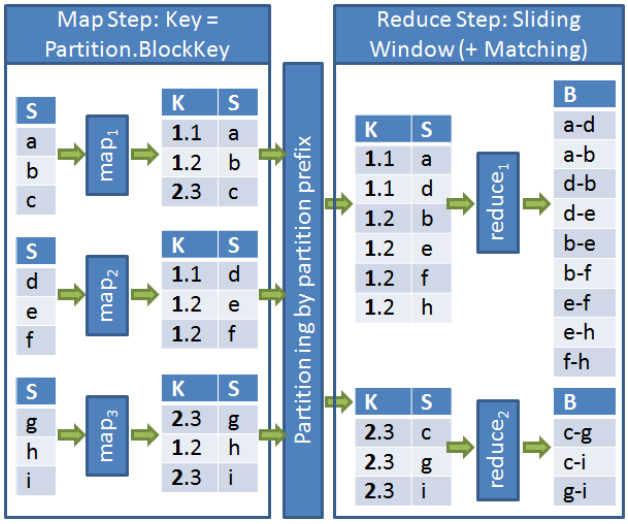

Figure 5: Example execution of sorted data partitioning with a composite key consisting of a blocking key and a partition prefix. The composite key ensures that the reduce partitions are ordered. If the sliding window approach ($w = 3$) is applied to both reduce partitions, it is only able to identify 12 out of the 15 SN correspondences (as shown in Figure 4). The pairs (f,c), (h,c), and (h,g) can not be found since the involved entities reside in different reduce partitions.

4.1 Sorted Reduce Partitions

We achieve sorted reduce partitions (SRP) by utilizing an appropriate user-defined function *part* for data redistribution among reduce tasks in the map phase. Data redistribution is based on the generated blocking key k, i.e., *part* is a function $part : k \rightarrow i$ with $1 \leq i \leq r$ and r is the number of reduce tasks. A monotonically increasing function *part* (i.e., $part(k_1) \geq part(k_2)$ if $k_1 \geq k_2$) ensures that all entities assigned to reduce task i have a smaller or equal blocking key than any entity processed by reduce task $i+1$.

The range of possible blocking key values is usually known beforehand for a given dataset because blocking keys are typically derived from numeric or textual attribute values. In practice simple range partitioning functions *part* may therefore be employed.

The execution of SRP is illustrated in Figure 5 for $m = 3$ map and $r = 2$ reduce tasks. It uses the same entities and

blocking keys as the example of Figure 4. In this example the function *part* is defined as follows: $part(k) = 1$ if $k \leq 2$, otherwise $part(k) = 2$. The map function first generates the blocking key k for each input entity and adds $part(k)$ as a prefix. In the example of Figure 5, the blocking key value for c is 3 and $part(k) = 2$. This results in a combined key value 2.3. The partitioning then distributes the (*key*,*value*) pairs according to the partition prefix of the key. For example, all keys starting with 2 are assigned to the second reduce task. Moreover, the input partitions for each reduce task are sorted by the (combined) key. Since all keys of reduce task i start with the same prefix i, the sorting of the keys is practically done based on the actual blocking key.

Afterwards the reduce task can run the sliding window algorithm and, thus, generates the correspondences of interest. Figure 5 illustrates the resulting correspondences as reduce output (B=Blocking). For entity resolution the reduce function will apply a matching approach to the correspondences. Reduce will therefore likely return a small subset of B. However, since we investigate in blocking techniques we leave B as output to allow for comparison with other approaches (see Section 4.2 and 4.3).

The sole use of SRP does not allow for comparing entities with a distance $\leq w$ that spread over different reduce tasks. For example, standard SN determines the correspondence (h,c) (see Figure 4) that can not be generated since h and c are assigned to different reduce tasks. For r reduce tasks and a window size w, SRP misses $(r-1) \cdot w \cdot (w-1)/2$ boundary correspondences. We therefore present two approaches, JobSN and RepSN, that build on SRP but are also able to deal with boundary entities.

4.2 JobSN: Sorted Neighborhood with additional MapReduce job

The JobSN approach utilizes SRP and employs a second MapReduce job afterwards that completes the SN result by generating the boundary correspondences. JobSN makes thereby use of the fact that MapReduce provides sorted partitions to the reduce tasks. A reduce task can therefore easily identify the first and the last $w-1$ entities during the sequential execution. Those entities have counterparts in neighboring partitions, i.e., the last $w-1$ entities of a reduce task relate to the first $w-1$ entities of the succeeding reduce task. In general, all reduce tasks output the first and last $w-1$ entities with the exception of the first and the last reduce task. The first (last) reduce task only returns the last (first) $w-1$ entities.

The pseudo-code for JobSN is shown in the appendix in Algorithm 1[3]. Figure 6 illustrates a JobSN execution example. It uses the same data of Figure 5. The map step of the first job is identical with SRP of Figure 5 and omitted in Figure 6. The reduce step is extended by an additional

[3] $p = 1$ for single-pass JobSN

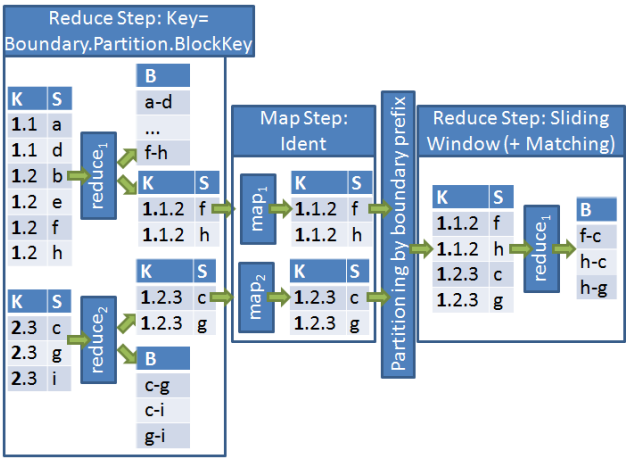

Figure 6: Example execution of SN with additional MapReduce job (JobSN, $w = 3$). The far left box is the reduce step of the first job. Its output is the input to the second MapReduce job.

output. Besides the list of blocking correspondences B, the reduce task also emits the first and last $w-1$ entities.

JobSN realizes the assignment of related boundary elements with an additional boundary prefix that specifies the boundary number. Since the last $w-1$ entities of reduce task $i < r$ refer to the i^{th} boundary, the keys of the last $w-1$ entities are prefixed with i. On the other hand, the first $w-1$ entities of the succeeding reduce task $i+1$ also relate to the i^{th} boundary. Therefore the keys of the first $w-1$ entities of reduce task $i > 1$ are prefixed with $i-1$. The first reduce task in the example of Figure 6 prefixes the last entities (f and h) with 1 and the second reduce task prefixes the first entities (c and g) with 1, too. Thereby the key reflects data lineage: The actual blocking key of entity c is 3 (see, e.g., Figure 4), it was assigned to reduce task number 2 during the SRP (Figure 5), and it is associated with boundary number 1 (Figure 6).

The second MapReduce job of JobSN is straightforward. The map functions leaves the input data unchanged. The map output is then redistributed to the reduce tasks based on the boundary prefix. The reduce function then applies the sliding window but filters correspondences that have already been determined in the first MapReduce job. For example, (f,h) does not appear in the output of the second job since this pair is already determined by SRP. As mentioned above, this knowledge is encoded in the lineage information of the key because those entities share the same partition number.

The JobSN approach generates the complete SN result at the expense of an additional MapReduce job. We expect the overhead for an additional job to be acceptable and we will evaluate JobSN's performance in Section 6.

4.3 RepSN: Sorted Neighborhood with entity replication

The RepSN approach aims to realize SN within a single MapReduce job. It extends SRP by the idea that each reduce task $i > 1$ needs to have the last $w-1$ entities of the preceding reduce task $i-1$ in front of its input. This would ensure that the boundary correspondences appear in the reduce task's output. However, the MapReduce paradigm is not designed for mutual data access between different reduce tasks. MapReduce only provides options for controlled data replication within the map function.

The RepSN approach therefore extends the original SRP map function so that map replicates an entity that should be send to both the respective reduce task and its successor. For all but the last reduce partition r, the map function thus identifies the $w-1$ entities with the highest blocking key k. It first outputs all entities and adds the identified boundary entities afterwards. Similar to SRP, an entity key is determined by the blocking key k plus a partition prefix $part(k)$. To distinguish between original entities and replicated boundary entities, RepSN adds an additional boundary prefix. For all original entities this boundary prefix is the same as the partition number, i.e., the composite key is $part(k).part(k).k$. The boundary prefix for replicated entities is the partition number of the succeeding reduce task, i.e., the composite key is $(part(k)+1).part(k).k$.

RepSN is described in the appendix in Algorithm 2[4]. Figure 7 illustrates an example execution of RepSN. The example employs $r = 2$ reduce tasks and window size $w = 3$. Therefore all map tasks identify the $w-1 = 2$ entities with the highest key of partition 1. The output of each map function is divided into two parts. The upper part (above the solid line) is equivalent to the regular map output of SRP. The only (technical) difference is that the partition prefix is duplicated. The lower part (framed by a dashed line) of the map output contains the replicated entities. Consider the second map function: All three entities (d, e, and f) are assigned to the partition 1 and e and f are replicated because they have the highest keys. The keys of the replicated data start with the succeeding partition 2. This ensures that e and f are send to both reduce task 1 and reduce task 2.

The map output is then redistributed to the reduce functions based on the boundary prefix. Furthermore, MapReduce provides a sorted list as input to the reduce functions. Due to the structure of the composite key, the replicated entities appear at the beginning of each reduce task input. Replicated entities share the same boundary prefix but have a smaller partition prefix. The reduce function then applies the sliding window approach but only returns correspondences involving at least one entity of the actual partition.

[4] $p = 1$ for single-pass RepSN

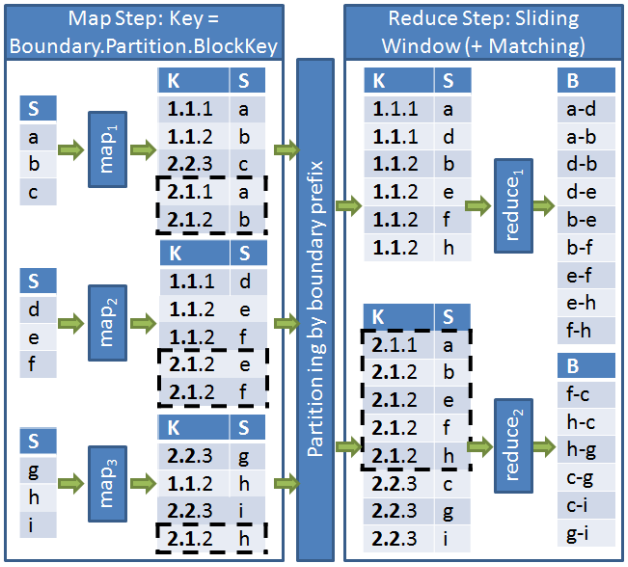

Figure 7: Example execution of sorted neighborhood with entity replication (RepSN, $w = 3$). Entities are replicated within in the map function. Replicated entities are framed by a dashed line.

In the example of Figure 7, input and output of the first reduce task are equivalent to SRP (see Figure 5). The second reduce task receives a larger input partition. It ignores all replicated entities but the $w-1 = 2$ highest (f and h). The output is the union of the corresponding SRP output and the corresponding boundary reduce output of JobSN.

RepSN allows for an entire sorted neighborhood computation within a single MapReduce job at the expense of some data replication. Since the MapReduce model does not provide any global data access[5] during the computation, it is not possible to identify only the necessary entities for processing the boundary elements. Rather each map function has to identify and replicate possibly relevant entities based on its local data. Each map task has to replicate $w-1$ entities for all but the last partition. The maximum number of replicated entities is therefore $m \cdot (r-1) \cdot (w-1)$. This number is independent from the size n of input entities and may therefore be comparatively small for large datasets. We will evaluate the overhead of data replication and data transfer in Section 6. In particular we will compare it against the JobSN overhead for scheduling and executing an additional MapReduce job.

[5] Hadoop as the most popular implementation MapReduce offers a so-called distributed cache. However, the primary purpose of this mechanism is to upfront copy read-only data (like files or archives) needed by the job to the particular nodes.

4.4 Multi-pass Sorted Neighborhood

Using a single blocking key may not sufficiently allow finding all duplicates especially with dirty input data. Furthermore, the utilized window size may have to be very large in order to identify most duplicates. For example, the window size should be at least as large as the highest number of entities with the same blocking key. However, large window sizes lead to numerous entity comparisons and thus limited performance. Multi-pass SN addresses these problems by employing multiple blocking keys and match passes and combining the duplicates identified in the different passes. Another advantage of this approach is that the individual passes can be done with relatively small windows sizes so that multi-pass SN can significantly improve both match effectiveness and efficiency [14].

A naïve approach to implement multi-pass SN runs one of the proposed MapReduce-based implementations p times and employs one of p different blocking key functions per pass. This approach requires scanning the input dataset p times and introduces additional overhead for executing multiple MapReduce jobs.

We therefore describe how our RepSN strategy can be extended so that multi-pass SN can be realized within a single MapReduce job (called MultiRepSN). The extension of JobSN is feasible analogously but we omit its description here due to space constraints. The key idea is that we add the pass number as a prefix to the single-pass RepSN composite map key. For each input entity and each pass $i \in \{1,\ldots,p\}$ map thus outputs a key-value pair with a composite map key of $i.part_i(k).part_i(k).k$ where k denotes the blocking key according to the blocking key function of pass i. For each pass, we employ a pass-specific partitioning function $part_i$ to enable an even data partitioning for the respective blocking key of the pass. Like in the single-pass RepSN approach, map identifies boundary elements for all passes and adds corresponding key-value pairs with a key of $i.(part_i(k)+1).part_i(k).k$ to the output.

The map output is partitioned among the reduce tasks by the boundary index and sorted by the entire key. Since the pass number is the first key component, the keys are sorted by the pass number first and then like in single-pass RepSN. The reduce task may therefore apply the sliding window approach as in RepSN by focusing on pairs of a specific pass.

We extend our single-pass example to two passes where the first pass is the same as before. The blocking key function of the second pass assigns a blocking key of 9 to entities a and d; all other entities have blocking key 8. The corresponding partitioning function distributes the two blocking keys to the two reduce tasks such that $part_2(k) = 1$ if $k = 8$ and $part_2(k) = 2$ if $k = 9$. The window size is set to $w_2 = 2$. Figure 8 illustrates the MultiRepSN approach for two passes. The first part of each map output is the same as in Figure 7 with an additional pass prefix of 1. The second

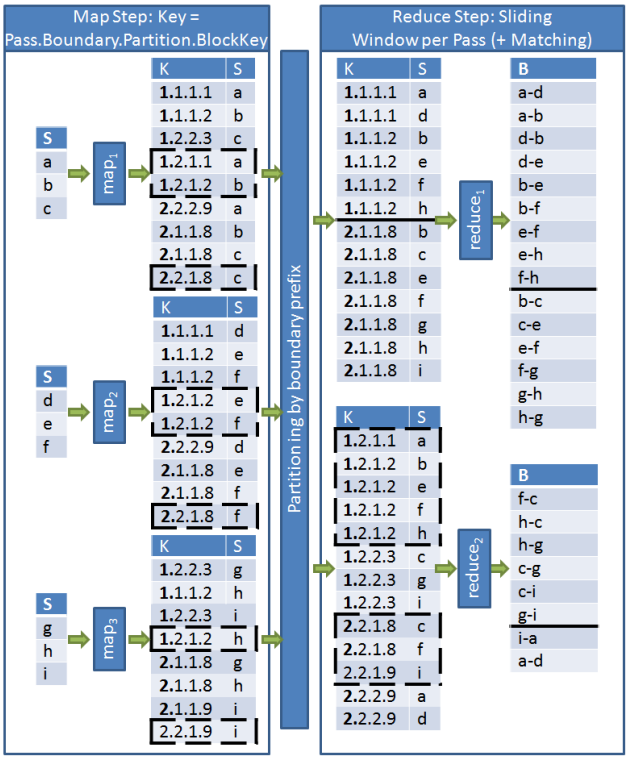

Figure 8: Example dataflow for MultiRepSN for two passes (window sizes $w_1 = 3$ and $w_2 = 2$).

pass entails the comparison of additional pairs thus helping to find additional duplicates.

5 Automatic partitioning for N-pass Sorted Neighborhood

The choice of a proper range partitioning function part is a critical step to enable evenly utilized reduce tasks and thus efficient SN processing. Finding a function that assigns a fairly equal amount of blocking entity pairs to each reduce task is a non-trivial task and usually requires an upfront data analysis. A partitioning function must thereby take into account not only the blocking key distribution but also the employed window size. The effort is further increased for multi-pass approaches because in each of the p passes different blocking key functions are applied resulting in different key ranges and key frequencies. Furthermore, different passes can employ different window sizes. A *manual partitioning* would therefore impose a very high configuration effort that should be avoided by an automatic approach.

The example of Figure 8 already illustrates the need for *automatic partitioning*. Although both employed partitioning functions (one for each pass) are fairly reasonable, the resulting reduce task workload is unbalanced. The first re-

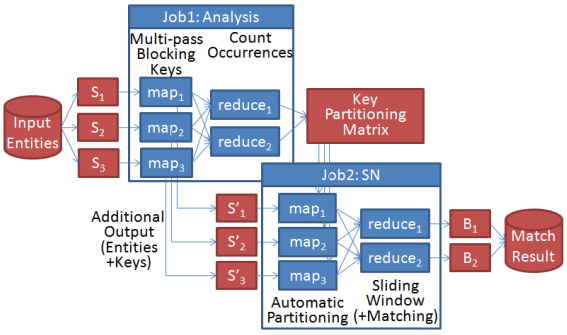

Figure 9: Schematic overview of MultiRepSN with global partitioning

duce task processes 15 pairs whereas the second reduce task has only 8 pairs to evaluate.

We propose the use of an additional preprocessing (analysis) MR job to achieve automatic partitioning and load balancing for both MapReduce-based SN strategies. Figure 9 illustrates the schematic workflow for MultiRepSN. Again, the presented techniques can be straightforwardly adopted to JobSN but we omit the details for brevity. Both MR jobs are based on the same number of map tasks and the same partitioning of the input data. The first (analysis) job calculates a so-called *key partitioning matrix* (*KPM*) that specifies the number of entities per key and pass separated by input partitions. The matrix is used by the map tasks of the actual SN approach in the second MR job to automatically identify an optimal partitioning function and, thus, tailor entity redistribution among reduce tasks.

5.1 KPM computation

The key partitioning matrix (KPM) specifies the number of entities of all blocking keys across m input partitions. Using MR for the the KPM computation is relatively simple. For each entity and each pass $i \in \{1,\ldots,p\}$ the map task determines the blocking key k and outputs a key-value with a composite key $i.k_i.j$ and the entity as the corresponding value. The last part of the composite key denotes the map task (input partition) index $j \in \{1,\ldots,m\}$. The key-value pairs are partitioned based on the blocking key component and sorted. The reduce task outputs the number of entities per map key, i.e., it determines the number of entities for each combination of pass, blocking key, and map partition.

Figure 10 illustrates the computation of the KPM for our two-pass example. For example, blocking key 3 is assigned to entities c, g, and i for the first pass. Since entity c is in the first map partition and g and i belong to the third partition, the KPM contains a key distribution of 1, 0, and 2 across the three map tasks for $p = 1$ and $k = 3$. As illustrated in Figure 9, map produces an additional output S_i per map partition that contains the original entities annotated with their

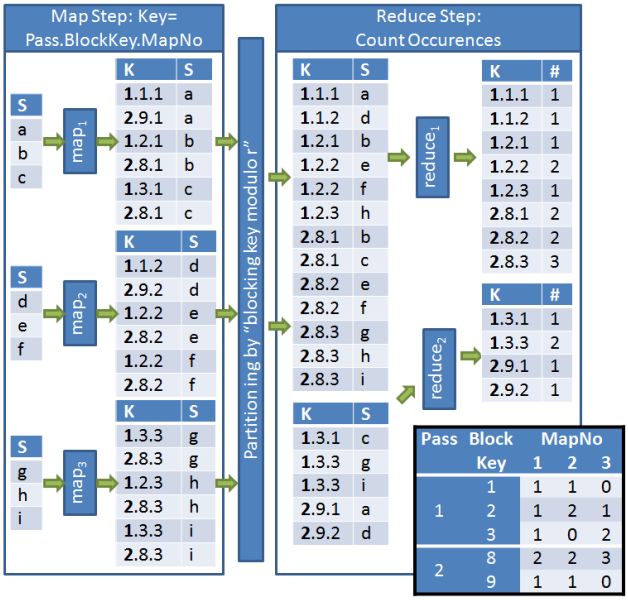

Figure 10: Example dataflow for computation of the key partitioning matrix (MR job 1 of Figure 9)

blocking keys and pass number. This output is not shown in Figure 10 to save space but used as input in the second MR job to avoid unnecessary parsing and key generation during the second job.

The KPM allows for a sorted enumeration of all entities as follows: Entities are ordered – in this sequence – by the pass number, blocking key, and map task (input partition) number. Entities of the same map task sharing the same pass number and blocking key are enumerated by their (arbitrary) order within the input partition. We can therefore assign a global position $pos \in \{1,\ldots,p \cdot n\}$ for each entity in every pass, i.e. p times. The quadruple (pass number, blocking key, map input partition, entity index) unambiguously determines the global position pos whereas the entity index refers to the n^{th} entity for a (pass, blocking key, map input partition) combination.

5.2 Automatic partitioning utilizing the KPM

In the following we describe how the RepSN strategy can make use of a KPM and the global enumeration order to achieve a balanced distribution of entity pairs to be evaluated by multi-pass SN and given window sizes. Each map tasks reads the KPM, i.e., the output of the preceding MR job, and calculates the overall number of pairs $N = \Sigma_{i=1}^{p}(n - \frac{w_i}{2}) \cdot (w_i - 1)$ and the average number of pairs per reduce task $N_\emptyset = \frac{N}{r}$.

The map task then uses the overall entity enumeration and determines for each reduce task j the entity with the smallest position pos_j so that the number of pairs for all

reduce tasks 1 to j would be greater or equal $j \cdot N_\emptyset$. Note that this does not mean that the map task "knows" the entity. Each map task still has only access to its input partition. However, it is sufficient that it can specify its position based on pass number, blocking key, map number, and position within the input partition.

The partitioning function is then defined as follows: $part(e) = j \Leftrightarrow pos_{j-1} \leq pos(e) \leq pos_j$ with $pos_0 = 1$. With the help of the KPM each map task also identifies the necessary entities for replication. Entities with positions between $pos_{j-1} - (w-1)$ and $pos_{j-1} - 1$ are subject to replication for each $j > 1$ as long as they belong to the same pass as entity with position pos_j. The automatic partitioning using a KPM thereby avoids unnecessary replication in contrast to manual partitioning that may generate unnecessary replicated entities (e.g., a, b, and e in the example of Figure 7) because each map task is unaware of the existence of entities in other input partitions. Algorithms 3 and 4 in the appendix show the pseudo code of the KPM computation and the automatic partitioning function that can be employed by the SN implementations.

Figure 11 shows an example of automatic partitioning for our running example. The $p = 2$ passes lead to $N = (9 - \frac{3}{2}) \cdot (3-1) + (9 - \frac{2}{2}) \cdot (2-1) = 23$ pairs for $n = 9$ entities and window sizes $w_1 = 3$ and $w_2 = 2$, respectively. For $r = 2$ reduce tasks the average pair number is $N_\emptyset = 11.5$. Since the first pass uses a larger window than the second pass, entities of the first pass are processed by both reduce tasks whereas the second pass is processed by the second reduce task only. Entity g of pass #1 is the last entity that needs to be sent to the first reduce task because all entities of pass #1 up to g (a, d, b, ..., c, g) account for 13 pairs whereas entities up to c (g's predecessor) account for 11 pairs only. All other entities, i of pass #1 and all entities of pass #2, are sent to the second reduce task. Due to window size $w_1 = 3$, g and its predecessor c are replicated and sent also to the second reduce task to ensure that all pairs involving i are covered. The resulting reduce task workload distribution is improved from 15/8 (Figure 8) to 13/10. The small difference to an optimal workload of 12/11 is due to the fact that entities usually take part in more than one pair. However, the divergence is almost smaller than or equal to $\max(w_i) - 1$ and, thus, acceptable compared to the total number of pairs. In other words, any two reduce tasks differ by no more than $2 \cdot (\max(w_i) - 1)$ pairs (see Appendix B for a proof).

6 Experiments

We conducted a set of experiments to evaluate the efficiency and effectiveness of the proposed approaches. After a description of the experimental setup we first compare single-pass JobSN and RepSN and study the scalability of our SN approaches. Afterwards we evaluate multi-pass SN and the proposed automatic partitioning approach. We finally study the trade-off between quality and execution time for single- and multi-pass SN.

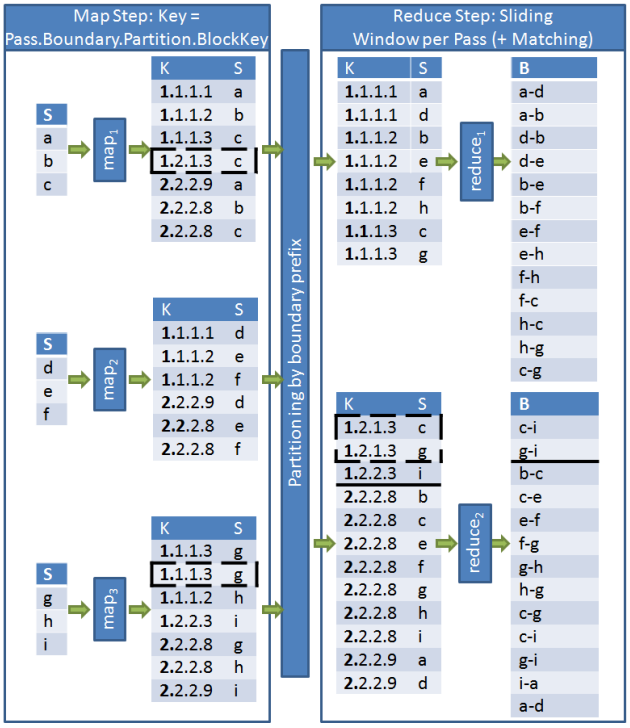

Figure 11: Example dataflow for MultiRepSN with 2 passes and global partitioning. The input is the additional output of job 1 (see Figure 9).

6.1 Experimental setup

We run our experiments on up to four nodes with two cores. Each node has an Intel(R) Core(TM)2 Duo E6750 2x2.66 GHz CPU, 4GB memory and runs a 64-bit Debian Linux OS with a Java 1.6 64-bit server JVM. On each node we run Hadoop 0.20.2. Following [23] we made the following changes to the Hadoop default configuration: We set the block size of the DFS to 128MB, allocated 1GB to each Hadoop daemon and 1GB virtual memory to each map and reduce task. Each node was configured to run at most two map and reduce tasks in parallel. Speculative execution was turned off. Both master daemons for managing the MapReduce jobs and the DFS run on a dedicated server. We used Hadoop's SequenceFileOutputFormat with native bzip2 block compression to serialize the output of map and reduce tasks that was further processed. Sequence files can hold binary key-value pairs what allowed us to directly access the i^{th} attribute value of an entity during matching in comparison to split a string at runtime.

For our experiments we use the same input dataset as in [23] that contains about 1.4 million publication records.

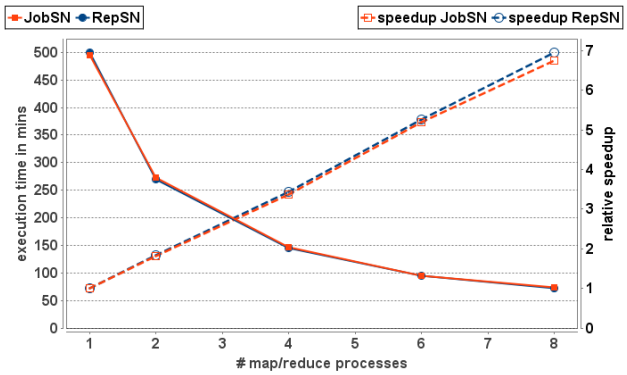

Figure 12: Comparison of the two Sorted Neighborhood implementations ($p = 1$, manual partitioning, $w = 1000$)

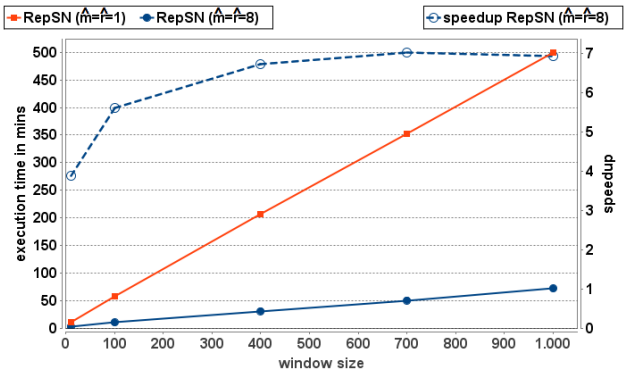

Figure 13: Runtime of RepSN (p=1, manual partitioning) for different window sizes w using $\hat{r}$=1 and $\hat{r}$=8 reduce processes, respectively.

To compare publications we determine the average similarity of two matchers (edit distance on title, TriGram on abstract). Pairs of entities with an average similarity score of at least 0.7 are regarded as matches. We apply an internal optimization by skipping the execution of the second matcher if the similarity after the execution of the first matcher is too low (i.e., < 0.4) to reach the combined similarity threshold. As a default blocking key for single-pass SN, we use the lower-cased first two letters of the publication title.

6.2 Comparison of RepSN and JobSN

We first evaluate the absolute runtime and the relative speedup of the two SN implementations for a fixed window size of 1000. To ensure comparability for different numbers of map and reduce tasks we apply the same manually defined partitioning function[6] in each experiment. It operates only on the blocking key, i.e., assigns all entities with the same blocking key to the same reduce task. The set of entities is divided into 10 partitions of preferably similar size. However, their sizes differ because on the one hand the MapReduce paradigm requires entities sharing the same blocking key to be processed within the same reduce task and on the other hand the partitions have to be sorted. The resulting 10 reduce tasks are executed by at most $\hat{r} = 8$ reduce processes allowing the matching of several small partitions while a large partition is processed.

Figure 12 shows execution times and speedup results for up to 8 map and 8 reduce processes for the two proposed implementations. The configuration with $\hat{m} = \hat{r} = 1$ refers to sequential execution on a single node, the one with $\hat{m} = \hat{r} = 2$ refers to the execution on a single node utilizing both cores and so on. The execution times scale almost linearly, for instance the execution time for RepSN could be reduced from approximately 8.3 to merely 1.2 hours. We observe a nearly linear speedup for the entire range of up to 4 nodes and 8 cores. The runtime of the different implementations differ only slightly. Differences can only be observed for a small amount of parallelism, i.e., RepSN was 5 minutes slower in the sequential case. Beginning with $\hat{m} = \hat{r} = 2$ RepSN completed slightly faster than JobSN due to the avoidance of a separate MR job for boundary comparisons. The reasons for the speedup values (about 7 for 8 cores) are caused by design and implementation choices of MapReduce/Hadoop to achieve fault tolerance, e.g., materialization of (intermediate) results between map and reduce.

[6]The space of possible blocking keys was partitioned into ten partitions utilizing the split points $a!, b, d, f, k, p, s, t, ti$. Therefore, an entity whose blocking key is less than or equal to $a!$ (e.g. 'a ') is assigned to the first reduce task.

6.3 Scalability according to window size

Due to the largely similar performance of RepSN and JobSN we focus on RepSN in our further experiments. Next, we evaluate the absolute runtime and the relative speedup of RepSN ($p = 1$) using varying window sizes from 10 and 1000 and compare the results for $\hat{r} = 1$ and $\hat{r} = 8$ reduce processes. We apply the same partitioning strategy as in our previous experiment. Figure 13 shows that the execution time increases linearly with the window size because the number of pairs also grows linearly with the window size. The speedup of parallel SN improves with larger window sizes from about 4 for small window sizes ($w = 10$) to 7 for larger window sizes ($w \geq 400$). For the latter the execution time is dominated by the pair-wise matching during the reduce phase whereas the additional MapReduce management overhead is more noticeable for smaller window sizes.

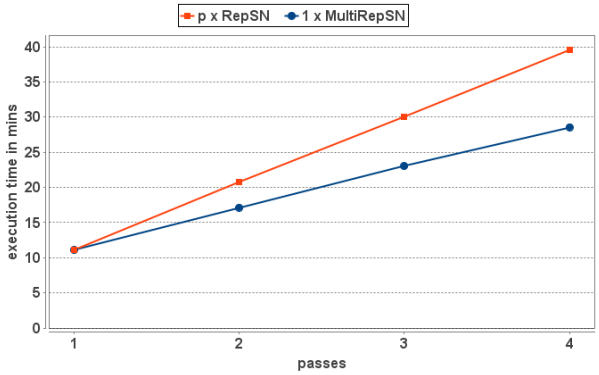

Figure 14: Comparison of the two approaches for multi-pass SN using RepSN ($w_i = 100$). $p\times$ RepSN executes single-pass RepSN p times whereas $1\times$ RepSN realizes multi-pass RepSN within one MapReduce job. Automatic partitioning is applied for both approaches.

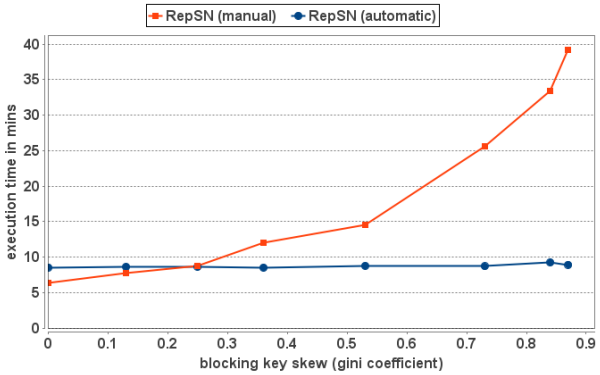

Figure 15: Influence of data skew for manual and automatic partitioning ($\hat{m} = \hat{r} = 8$, $w = 100$)

6.4 Multi-pass Sorted Neighborhood

We compare our multi-pass MultiRepSN strategy against the naïve strategy of a repeated execution of single-pass RepSN. To eliminate the influence of data skew (see Section 6.5) we apply automatic partitioning for both approaches. Figure 14 shows the result for 1 to 4 passes and a fixed window size of 100. In addition to the default single-pass blocking we utilize three blocking functions: last name of first author, first letter of last names of all authors, and first letter of publication title plus first author name.

We observe that the runtime grows linearly with the number of passes because the number of processed pairs is linear to the number of entities and passes. Furthermore, the automatic partitioning achieves that all reduce tasks receive about the same number of pairs. The $p\times$ RepSN strategy mainly suffers from the fact that the input data needs to be read and parsed p times while the additional overhead of multiple MR job is relatively low. MultiRepSN, on the other hand, avoids the p-fold input reading and clearly outperforms the naïve strategy. The execution time improvements increase with more passes, e.g., from about 17% for two to 28% for four passes.

6.5 Automatic partitioning and data skew

We study the influence of data skew on runtime. To this end, we control the degree of data skew by modifying the blocking function and generating block distributions that follow an exponential distribution. Given a fixed number of blocking keys b=8, we set the number of entities in the i^{th} block as proportional to $e^{-s\cdot i}$. We thereby use the skew factor $s \geq 0$ to control the degree of data skew. To quantify the inequality of the blocking key distribution in the dataset we utilize the Gini coefficient $g = \frac{2\cdot\sum_{i=1}^{b} i\cdot K_i}{b\cdot\sum_{i=1}^{b} K_i} - \frac{b+1}{b}$, whereas K_i is the number of entities in block i and $K_i \leq K_{i+1}$ for $i \in \{1,\ldots,b-1\}$. A value of 0 expresses total equality ($s = 0$) and a value of 1 maximal inequality ($s \to \infty$).

We run our experiments on 4 nodes (8 map and 8 reduce processes) with the same window size $w_i = 100$ for each pass i. We apply the RepSN strategy with and without automatic partitioning. In the latter case, we employ a simple hand-crafted partitioning function that assigns the i^{th} out of $b = 8$ blocks to the i^{th} out of $r = 8$ reduce tasks. Figure 15 shows the corresponding run times.

As expected, the manual partitioning is susceptible to data skew. Significant data skew leads to an unbalanced reduce task workload which deteriorates the overall execution time that is determined by the last finishing reduce task. In contrast, the automatic partitioning is able to achieve a balanced reduce task workload. Since the number of pairs is *not* affected by data skew, Figure 15 shows an almost constant run time for all data sets. The overhead of KPM computation is approximately 2.5 minutes which is why manual partitioning is more efficient for relatively unskewed data. As shown in Figure 15 the utilization of automatic partitioning is beneficial for Gini coefficients > 0.25[7]. Blocking keys of real-world datasets are typically very unevenly distributed. We argue that due to the large potential for runtime improvement and the relatively small costs of analyzing the data up-front, an automatic partitioning approach should always be preferred compared to a manual adjustment of split points for the blocking keys. Moreover, as demonstrated, a strict partitioning by blocking key can not distribute the processing of very large blocks to multiple reduce tasks.

[7] As a reference, the carefully chosen manual partition function has a Gini coefficient of 0.13. A simple partitioning strategy for $r = 8$ with the blocking key intervall split points c, f, i, l, o, r, u leads to a gini Gini coefficient of 0.32.

6.6 Quality vs. execution time

We finally evaluate the trade-off between match quality and execution time. The dataset used so far could not be applied for this experiment due to the absence of a gold standard for evaluating match quality. We thus used another publication dataset (based on [19]) that contains about 8,000 records and provides a perfect match result needed to compute precision, recall, and F-Measure. We employ the single-pass RepSN approach with a window size $w = 1000$ and compare it against three MultiRepSN strategies (p=2) with window sizes $w_1 = w_2 = 500$, $w_1 = w_2 = 200$ and $w_1 = w_2 = 100$, respectively. Blocking keys are the first letter of the first author's name (pass #1) and the first letter of the publication title (pass #2), respectively. Two entities are considered to match if their titles have a trigram similarity greater or equal than 0.75. As before, we run our experiments on 4 nodes ($\hat{m} = \hat{r} = 8$) using automatic partitioning.

The upper part of Figure 16 shows the observed execution times along with the number of comparisons. The reduction rate specifies the ratio of saved comparisons relative to the Cartesian product of $31.7 \cdot 10^6$ comparisons. The lower part of Figure 16 depicts the quality of the obtained match result for all four strategies. Precision, recall, and F-Measure are computed separately for all pairs that have been directly identified by SN (Pairs) and all pairs of the transitive closures (TC). Due to the small dataset size, the transitive closures of matching entity pairs were calculated as a post-processing step without using MapReduce.

Both single-pass ($w = 1,000$) and multi-pass ($w = 500$) result in a similar number of comparisons and, thus, have a similar execution time. In both cases the evaluation of the transitive closure of matching pairs is highly beneficial since it leads to additional matches and thus improved recall. The multi-pass approach achieves a significantly better F-Measure than single-pass SN mainly due to improved recall. Hence, the consideration of two blocking keys helps to find many more matches despite the smaller window sizes.

urther reducing the window size for multi-pass SN to $w = 200$ significantly improves execution time (by about 25%) compared to the single-pass configuration. Still when applying the transitive closure a significantly better F-measure is achieved for $w = 200$. Even for $w = 100$ (execution time improvement by about 40%) a similar F-Measure is achieved compared to the single-pass configuration.

The shown results confirm that multi-pass SN is able to achieve high match quality even with smaller window sizes and that it can outperform single-pass SN not only in match quality but also in efficiency due to reduced window sizes.

	Single-pass		Multi-pass (p=2)					
Window size	w=1,000		w_1=w_2=500		w_1=w_2=200		w_1=w_2=100	
#Comparisons	≈7.5·10⁶		≈7.7·10⁶		≈3.1·10⁶		≈1.6·10⁶	
Reduction Ratio	76.5%		75.7%		90.3%		95.1%	
Execution time [in s]	110		109		81		69	
	Pairs	TC	Pairs	TC	Pairs	TC	Pairs	TC
Precision	91.7%	90.7%	88.8%	82.9%	88.1%	82.7%	87.1%	81.8%
Recall	62.0%	71.0%	75.5%	88.5%	68.7%	85.9%	58.5%	77.2%
F-Measure	74.0%	79.6%	81.6%	85.6%	77.2%	84.3%	70.0%	79.5%

Figure 16: Comparison of quality and execution time for single- and multi-pass RepSN using different window sizes. (Pairs=identified match pairs; TC=transitive closure of all pairs)

7 Related work

Entity resolution is a very active research topic and many approaches have been proposed and evaluated as described in recent surveys [12, 18]. Surprisingly, there are only a few approaches that consider parallel entity resolution. First ideas for parallel matching were described in the Febrl system [7]. The authors show how the match computation can be parallelized among available cores on a single node. Parallel evaluation of the Cartesian product of two sources considering the three input cases (clean-clean, clean-dirty, dirty-dirty) is described in [15].

[16] proposes a generic model for parallel processing of complex match strategies that may contain several matchers. The parallel processing is based on general partitioning strategies that take memory and load balancing requirements into account. Compared to this work [16] allows the execution of a match workflow on the Cartesian product of input entities. This is done by partitioning the set of input entities and generating match tasks for each pair of partitions. A match task is then assigned to any idle node in a distributed match infrastructure with a central master node. The advantage of this approach is the high flexibility for scheduling match tasks and thus for dynamic load balancing. The disadvantage is that only the matching itself is executed in parallel. Blocking is done upfront on the master node. Furthermore in this work we rely on an widely used parallel processing framework that hides the details of parallelism and therefore is less error-prone.

We are only aware of one previous approach for parallel entity resolution on a cloud infrastructure [23]. The authors do not investigate Sorted Neighborhood blocking but show how a single token-based string similarity function can be realized with MapReduce. The approach is based on a complex workflow consisting of several MapReduce jobs. This approach suffers from similar load balancing problems as observed in Section 6.5 because all entities that

share a frequent token are compared by one reduce task[8]. In contrast to our Sorted Neighborhood approach large partitions for frequent tokens that do not fit into memory must be handled separately. This is because all entities that contain a specific token have to be compared with each other instead of comparing only entities with a maximum distance of less than w. Compared to [23], we are not limited to a specific similarity function but can apply a complex match strategy for each pair of entities within a window.

8 Conclusions and outlook

We have shown how single- and multi-pass Sorted Neighborhood blocking can be efficiently parallelized with MapReduce. We proposed two single-pass MapReduce-based SN implementations and demonstrated their high efficiency and scalability in an evaluation using real-world datasets. The proposed multi-pass implementation avoids reading and parsing the input data several times for different passes and is thus more efficient than a naïve implementation of repeated single passes. We also proposed a highly efficient approach for automatic data partitioning and load balancing that supports multi-pass SN with different window sizes per pass.

While we could effectively utilize MapReduce and its implementation Hadoop, even higher performance was prevented by some of their limitations such as insufficient support for pipelining intermediate data between map and reduce jobs. In future work, we plan to further investigate cloud-based entity resolution for other blocking and match techniques and compare them with the proposed SN approaches.

References

[1] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. rep., EECS Department, University of California, Berkeley (2009)

[2] Batini, C., Scannapieco, M.: Data Quality: Concepts, Methodologies and Techniques. Data-Centric Systems and Applications. Springer (2006)

[3] Baxter, R., Christen, P., Churches, T.: A comparison of fast blocking methods for record linkage. In: ACM SIGKDD, vol. 3, pp. 25–27 (2003)

[4] Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: KDD, pp. 39–48 (2003)

[8] The authors could slightly reduce the data skew by redistributing data based on the infrequent prefix tokens of a record's attribute value.

[5] Borthakur, D.: The hadoop distributed file system: Architecture and design. Hadoop Project Website (2007)

[6] Christen, P.: Febrl -: an open source data cleaning, deduplication and record linkage system with a graphical user interface. In: KDD, pp. 1065–1068 (2008)

[7] Christen, P., Churches, T., Hegland, M.: Febrl - a parallel open source data linkage system. In: PAKDD, pp. 638–647 (2004)

[8] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)

[9] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

[10] DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)

[11] DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical skew handling in parallel joins. In: VLDB, pp. 27–40 (1992)

[12] Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007)

[13] Foundation, A.S.: Hadoop. `http://hadoop.apache.org/mapreduce/` (2006)

[14] Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: SIGMOD Conference, pp. 127–138 (1995)

[15] Kim, H.S., Lee, D.: Parallel linkage. In: CIKM, pp. 283–292 (2007)

[16] Kirsten, T., Kolb, L., Hartung, M., Gross, A., Köpcke, H., Rahm, E.: Data partitioning for parallel entity matching. In: 8th International Workshop on Quality in Databases (2010)

[17] Kolb, L., Thor, A., Rahm, E.: Parallel sorted neighborhood blocking with mapreduce. In: BTW, pp. 45–64 (2011)

[18] Köpcke, H., Rahm, E.: Frameworks for entity matching: A comparison. Data Knowl. Eng. **69**(2), 197–210 (2010)

[19] Köpcke, H., Thor, A., Rahm, E.: Evaluation of entity resolution approaches on real-world match problems. In: VLDB, pp. 484–493 (2010)

[20] Köpcke, H., Thor, A., Rahm, E.: Learning-based approaches for matching web data entities. IEEE Internet Computing **14**, 23–31 (2010)

[21] Lin, J., Dyer, C.: Data-intensive text processing with mapreduce. Synthesis Lectures on Human Language Technologies **3**(1), 1–177 (2010)

[22] Rahm, E., Do, H.H.: Data cleaning: Problems and current approaches. IEEE Data Eng. Bull. **23**(4), 3–13 (2000)

[23] Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: SIGMOD Conference, pp. 495–506 (2010)

A Algorithms

Algorithm 1 and Algorithm 2 show the pseudo-code for the two proposed Sorted Neighborhood implementations JobSN and RepSN. Using $p = 1$ corresponds to the single-pass version of JobSN and RepSN, introduced in sections 4.2 and 4.3.

The pseudo-code for KPM computation and automatic partitioning is shown in Algorithm 3 and 4. With the global knowledge encoded in the KPM, for RepSN, the number of replicated entities can be reduced from $p \cdot m(r-1)(w-1)$ to $p(r-1)(w-1)$. For simplicity this is omitted here. The map tasks read the KPM at initialization time. It is not required that each map task holds the full KPM in memory. First of all, lines whose blocking keys do not occur in a map tasks input partition can be omitted. Secondly, a map task need not keep all columns in memory. For each pass and each (relevant) blocking key it is sufficient to store the overall sum of entities in previous map input partitions (Algorithm 4 Lines 10-19). This number is incremented every time an entity with this blocking key is processed for this pass (Line 31). Furthermore it would be possible to materialize the KPM to disk. For instance Hadoop's *MapFileOutputFormat* provides a possibility to turn the output of Algorithm 3 into a persistent Map-like data structure including an index for fast access. This output can be distributed to the map tasks using Haddop's *Distributed Cache* allowing local data access.

The setup and teardown functions *map_configure* and *map_close* are automatically invoked by Hadoop before/after a map task is executed (the same counts for reduce tasks). Map/reduce tasks can buffer data in memory to access shared data among several map/reduce tasks. A key of the form $x.y$ denotes a key composed of x and y. Composed keys are compared component-wise. The comments indicate which parts of the composite keys are used for map-side repartitioning and reduce-side sorting and grouping of entities.

The function *match(e1, e2)* compares two entities by applying an arbitrary matching strategy and outputs matching entity pairs. The partition to which an entity with blocking key k is assigned during pass i is determined by a generic pass-specific function *getPartition(i, k, r)* that can be either manually or automatically defined. Algorithms 1 and 3 use a function *additionalOut(key, value)* that writes key-value pairs to the distributed filesystem next to the regular map output in a binary format. These key-value pairs are later read by the map function of another MapReduce job in order to avoid multiple parsing and processing of input data. By prohibiting splitting of input files it is ensured that the second MapReduce job has the same number of map tasks than the previous one.

B Proof

We prove that the automatic partitioning using a KPM leads to a balanced workload across all reduce tasks such that the maximal difference between any two reduce tasks is $2 \cdot (\max(w_i) - 1)$ comparisons.

Let R_j be the number of comparisons assigned to reduce task j. The maximal window size is denoted as W and the average number of pairs per reduce task is N. The automatic partitioning (see Section 5.2) ensures that that the number of pairs for all reduce tasks 1 to j would be greater or equal $j \cdot N$. Therefore

$$R_1 + \ldots + R_{j-1} \geq (j-1) \cdot N \quad (1)$$
$$R_1 + \ldots + R_{j-1} + R_j \geq j \cdot N \quad (2)$$

Adding an entity e to a reduce tasks leads to up to $W-1$ additional pairs because e needs to be compared its $w_i - 1$ predecessors in pass i. Since the partitioning assigns the minimal range of entities that satisfies inequation (2) we can specify upper bounds:

$$R_1 + \ldots + R_{j-1} < (j-1) \cdot N + W - 1 \quad (3)$$
$$R_1 + \ldots + R_{j-1} + R_j < j \cdot N + W - 1 \quad (4)$$

We then conclude

$$\Rightarrow \quad (j-1) \cdot N \leq R_1 + \ldots + R_{j-1} < (j-1) \cdot N + W - 1 \quad (5)$$
$$\Rightarrow \quad j \cdot N \leq R_1 + \ldots + R_{j-1} + R_j < j \cdot N + W - 1 \quad (6)$$
$$(6) - (5)$$
$$\Rightarrow \quad N - (W-1) \leq R_j \leq N + W - 1 \quad (7)$$

Inequation 7 proves that the number of pairs for any reduce task j is between $N-(W-1)$ and $N+(W-1)$. Therefore two reduce tasks do not differ by more than $2 \cdot (W-1)$ pairs. □

Algorithm 1: Multi-pass JobSN

```
1  // --- Phase 1 ---
2  map_configure(jobConf)
3      p ← getNumberOfPasses(jobConf);
4      reduceTasks ← jobConf.numReduceTasks();
5  map(key_in=unused, value_in=entity)
6      for i ← 1 to p do
7          blockKey ← generateBlockingKeyForPass(i, entity);
8          partition ← getPartition(i, blockKey, reduceTasks);
9          // part(pass.partition.blockKey)= partition
10         output(key_tmp=i.partition.blockKey, value_tmp=entity);
11 reduce_configure(jobConf)
12     p ← getNumberOfPasses(jobConf);
13     lastPass ← -1;
14     windowSize ← getWindowSize(jobConf);
15     reduceTasks ← jobConf.numReduceTasks();
16     queue ← [];
17     for i ← 1 to p do
18         top_i ← [];
19         bottom_i ← [];
20 // order and group by composed key
21 reduce(key_tmp=i.partition.blockKey, list(value_tmp)=list(entity))
22     if i ≠lastPass then
23         queue ← [];
24         lastPass ← i;
25     foreach entity ∈ list(value_tmp) do
26         if partition>1 and top_i.size()<windowSize-1 then
27             top_i.addLast((partition, blockKey, entity));
28         if partition<reduceTasks then
29             if bottom_i.size() = windowSize-1 then
30                 bottom_i.removeFirst();
31             bottom_i.addLast((partition, blockKey, entity));
32         foreach e ∈ queue do
33             match(e, entity);
34         queue.addLast(entity);
35         if queue.size() = windowSize then
36             queue.removeFirst();
37 reduce_close()
38     for i ← 1 to p do
39         foreach (partition, blockKey, entity) ∈ top_i do
40             additionalOut(key_out=i.(partition-1).partition.blockKey,
41               value_out= entity);
42         foreach (partition, blockKey, entity) ∈ bottom_i do
43             additionalOut(key_out=i.partition.partition.blockKey,
44               value_out= entity);
45 // --- Phase 2 ---
46 // Read additional reduce output of phase 1
47 map(key_in=i.boundary.partition.blockKey, value_in=entity)
48     // part(pass.boundary.partition.blockKey)=
49     //     hash(pass, partition) % reduceTasks
50     output(key_tmp=i.boundary.partition.blockKey, value_tmp=entity);
51 reduce_configure(jobConf)
52     lastPass ← -1;
53     lastBoundary ← -1;
54     windowSize ← getWindowSize(jobConf);
55     queue ← [];
56 // order and group by composed key
57 reduce(key_tmp=i.boundary.partition.blockKey, list(value_tmp)=list(entity))
58     if i ≠ lastPass or boundary ≠ lastBoundary then
59         lastPass ← i;
60         lastBoundary ← boundary;
61         reduce_close();
62         queue ← [];
63     foreach entity ∈ list(value_tmp) do
64         queue.addLast((partition, entity));
65 reduce_close()
66     // Match all entites in queue with a distance of at
67     // most windowSize-1 and different partition indexes
```

Algorithm 2: Multi-pass RepSN

```
1  map_configure(jobConf)
2      p ← getNumberOfPasses(jobConf);
3      reduceTasks ← jobConf.numReduceTasks();
4      windowSize ← getWindowSize(jobConf);
5      // list of the entities with the w-1 highest blocking
6      // keys for each pass and each reduce task<r
7      for i ← 1 to p do
8          for j ← 1 to reduceTasks-1 do
9              rep^i_j ← [];
10 map(key_in=unused, value_in=entity)
11     for i ← 1 to p do
12         blockKey ← generateBlockingKeyForPass(i, entity);
13         j ← getPartition(i, blockKey, reduceTasks);
14         if j < reduceTasks then
15             if rep^i_j.size()<windowSize-1 then
16                 rep^i_j.add((blockKey, entity));
17                 rep^i_j.sort() ;    /* Sort by first component */
18             else if blockKey > rep^i_j.getFirst().firstComponent() then
19                 rep^i_j.removeFirst();
20                 rep^i_j.add((blockKey, entity));
21                 rep^i_j.sort() ;    /* Sort by first component */
22         // part(pass.boundary.partition.blockKey)
23         //     = boundary
24         output(key_tmp=i.j.j.blockKey, value_tmp=entity);
25 map_close
26     for i ← 1 to p do
27         for j ← 1 to reduceTasks-1 do
28             foreach (blockKey, entity) ∈ rep^i_j do
29                 // Adjust bound to j+1 to assign repli-
30                 // cated entities to next reduce task
31                 output(key_tmp=i.(j+1).j.blockKey, value_tmp=entity);
32 reduce_configure(jobConf)
33     p ← getNumberOfPasses(jobConf);
34     lastPass ← -1;
35     windowSize ← getWindowSize(jobConf);
36     queue ← [];
37 // order and group by composed key
38 reduce(key_tmp=i.boundary.partition.blockKey, list(value_tmp)=list(entity))
39     if i ≠ lastPass then
40         queue ← [];
41         lastPass ← i;
42     foreach entity ∈ list(value_tmp) do
43         if boundary = partition then
44             // no replicated entity
45             foreach e ∈ queue do
46                 match(e, entity);
47         queue.addLast(entity);
48         if queue.size() = windowSize then
49             queue.removeFirst();
```

Algorithm 3: Analysis job for KPM computation

```
1  map_configure(jobConf)
2      p ← getNumberOfPasses(jobConf);
3      partition ← getMapTaskIndex(jobConf);
4  map(key_in=unused, value_in=entity)
5      blockingKeys ← "";
6      for i ← 1 to p do
7          blockKey ← generateBlockingKeyForPass(i, entity);
8          blockingKeys ← blockingKeys + "\t" + blockKey;
9          // part(pass.blockingKey.partition)=
10         //     hash(pass, blockingKey) % reduceTasks
11         output(key_tmp=i.blockKey.partition, value_tmp=1);
12     additionalOut(key_out=blockingKeys, value_out=entity);
13 // Order and group by composed key
14 reduce(key_tmp=pass.blockKey.partition, list(value_tmp)=list(number))
15     sum ← 0;
16     foreach number in list(value_tmp) do
17         sum ← sum + number;
18     out ← pass + "\t" + blockKey + "\t" + partition + "\t" + sum;
19     output (key_out=unused, value_out=out);
```

Algorithm 4: AutoPartitioner utilizing the KPM

```
1  configure(jobConf, mapTasks, reduceTasks, p, partition)
2      KPM ← readKPM(jobConf);
3      n ← 0;
4      foreach blockKey ∈ KPM.getPass(1) do
5          for i ← 1 to mapTasks do
6              n ← n + KPM.getPass(1).getKey(blockKey).getPart(i);
7      // Global index of next entity of pass i and block j
8      // for this map task.  This map contains at most one
9      // entry per (pass,blockKey) pair
10     . entityIndexes ← empty map;
11     for i ← 1 to p do
12         entityIndex ← 1;
13         foreach blockKey ∈ KPM.getPass(i).getKeysSorted() do
14             for j ← 1 to mapTasks do
15                 if j = partition then
16                     entityIndexes.put((i, blockKey),
17                       (i-1)·n + entityIndex);
18                 entityIndex ← entityIndex +
19                   KPM.getPass(i).getBlock(blockKey).getPart(j);
20     // Pass-aware and window size-aware reduce task
21     // assignment for each entity, targeting an equal
22     // number of pairs per reduce task
23     splitPoints ← [];
24     N ← Σ_{i=1}^{p} (n − w_i/2) · (w_i − 1) /* w_i:  window size for pass i  */
25     N_∅ ← ⌈N/reduceTasks⌉;
26     pairsLeft ← N_∅;
27     offset ← 0;
28     for i ← 1 to p do
29         entitiesLeft ← n;
30         while entitiesLeft > 0 do
31             entitiesThatFit ← min{pairsLeft/w_i + w_i/2, entitiesLeft};
32             offset ← offset + entitiesThatFit;
33             splitPoints.addLast(offset);
34             entitiesLeft ← entitiesLeft - entitiesThatFit;
35             pairsThatFit ← (entitiesThatFit − w_i/2) · (w_i − 1);
36             pairsLeft ← pairsLeft - pairsThatFit;
37             if pairsLeft ≤ 0 then
38                 pairsLeft ← N_∅;
39 getPartition(pass, blockKey, reduceTasks)
40     entityIndex ← entityIndexes.get((pass, blockKey));
41     // adjust entity index for next call
42     entityIndexes.get((pass, blockKey)).put(entityIndex + 1);
43     for i ← 1 to splitPoints.size() do
44         if entityIndex ≤ splitPoints.get(i) then
45             return i;
```