Dedoop: Efficient Deduplication with Hadoop

Lars Kolb
Database Group
University of Leipzig
kolb@informatik.uni-leipzig.de

Andreas Thor
Database Group
University of Leipzig
thor@informatik.uni-leipzig.de

Erhard Rahm
Database Group
University of Leipzig
rahm@informatik.uni-leipzig.de

ABSTRACT

We demonstrate a powerful and easy-to-use tool called Dedoop (Deduplication with Hadoop) for MapReduce-based entity resolution (ER) of large datasets. Dedoop supports a browser-based specification of complex ER workflows including blocking and matching steps as well as the optional use of machine learning for the automatic generation of match classifiers. Specified workflows are automatically translated into MapReduce jobs for parallel execution on different Hadoop clusters. To achieve high performance Dedoop supports several advanced load balancing strategies.

1. INTRODUCTION

Entity resolution (ER) (also known as deduplication or object matching), is the task of identifying entities referring to the same real-world object [5]. It is a pervasive problem and 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. ER techniques usually compare pairs of entities by evaluating multiple similarity measures to make effective match decisions. As a consequence, ER is an expensive process that can take several hours or even days for large datasets [6]. A common approach to improve efficiency is to reduce the search space by adopting so-called blocking techniques [2]. For example, standard blocking utilizes a blocking key on the values of one or several entity attributes to partition the input data into multiple partitions (called blocks) and restrict the subsequent matching to entities of the same block. However, ER remains a costly process and, thus, is an ideal problem to be solved in parallel on cloud infrastructures.

We present Dedoop (Deduplication with Hadoop), an ER framework based on MapReduce (MR). The MR programming model is well suited for ER since the pair-wise similarity computation can be executed in parallel. The utilization of a cloud infrastructure significantly speeds up ER programs and, thus, has several advantages. First, manual tuning of ER parameters is facilitated since ER results can be quickly generated and evaluated. Second, the reduced execution times for large data sets speed up common data management processes, e.g., ETL programs for data warehouses. The highlights of Dedoop are as follows:

- Dedoop lets users easily specify advanced ER workflows in a web browser. Users can thereby choose from a rich toolset of common ER components (e.g., blocking techniques, similarity functions, etc.) including machine learning for automatically building match classifiers.
- Dedoop automatically transforms the workflow definition into an executable MapReduce workflow. The ER results and the workload of all cluster nodes can be visualized afterwards.
- Dedoop provides several load balancing strategies in combination with its blocking techniques to achieve balanced workloads across all employed nodes of the cluster. It is also able to avoid unnecessary entity pair comparisons that result from the utilization of multiple blocking keys.

In the following, we give a brief overview of Dedoop's architecture and implementation (Section 2) and provide details about Dedoop's techniques for efficient cluster utilization (Section 3). Finally, we present a detailed demonstration scenario (Section 4). More information can be found on Dedoop's project website [1].

2. OVERVIEW OF DEDOOP

Dedoop is an MR-based framework for ER and is tailored to the general ER workflow pattern as shown in the upper layer of Figure 1. The input is two entity sets (R and S). The output is all pairs $M \subseteq R \times S$ that are considered to match. The ER workflow consists of three consecutive steps: blocking, similarity computation, and the actual match decision (based on the computed similarity values). The latter can be based on a classifier that has been trained by a machine learning algorithms using training data (T).

Dedoop transforms a given ER workflow into a sequence of up to 3 MR jobs (classifier training, data analysis, and blocking-based matching) that can be executed on a Hadoop cluster (lower layer of Figure 1). To this end, Dedoop accesses an extensible ER component library (middle layer of Figure 1). In the following we describe all three MR jobs and the related Dedoop components.

Blocking-based Matching: Dedoop provides several MR-based implementations for blocking-based matching, e.g., Standard Blocking and Sorted Neighborhood (SN). All blocking techniques can employ several blocking key generators, e.g., concatenate the first k characters of certain

Proceedings of the VLDB Endowment, Vol. 5, No. 12

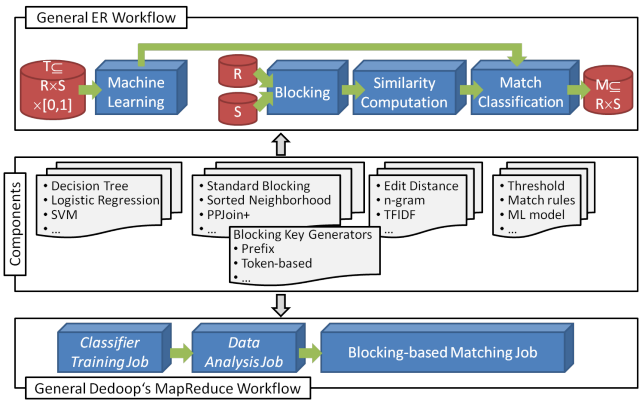

Figure 1: Overview of Dedoop.

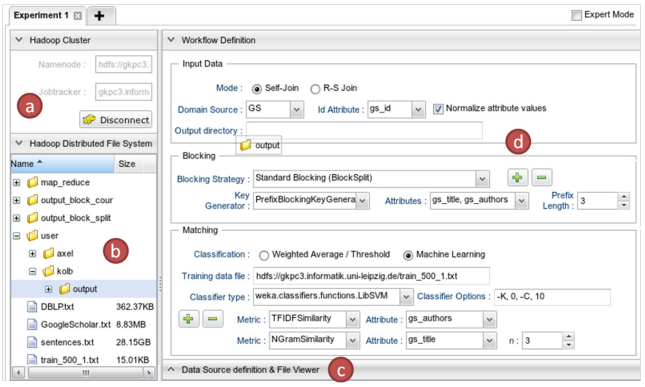

Figure 2: Dedoop's ER workflow configuration.

attribute values. For example, Standard Blocking [3] is realized within the map phase. 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 partitioning operates on the blocking keys and distributes 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 similarities for all entity pairs within its block.

For Sorted Neighborhood, the map function determines the blocking key for each input entity, similar to Standard blocking. Since SN assumes an ordered list of all entities based on their blocking keys, the partitioning between map and reduce phase preserves this order by a specific range partitioning function. The reduce tasks implement the sliding window approach for each reduce partition. To allow for comparing entities within the sliding window distance that spread over different reduce tasks, one of our approaches, RepSN [4], extends the original map function so that map replicates entities close to partition boundaries and sends them both the respective reduce task and its successor.

For similarity computation, the reduce phase executes the specified matchers from Dedoop's extensible library that include common string metrics such as TF/IDF or n-gram. For the final match decision Dedoop provides machine learning-based and threshold-based match classification. The blocking-based matching is mandatory for each generated MR workflow and by far the most time-consuming job.

Data Analysis (optional): Blocking-based approaches realize similarity computation in the reduce phase. Simple strategies are therefore highly vulnerable to data skew due to the quadratic complexity of costly entity comparisons per block. As a consequence, the execution time is often dominated by a single or a few reduce tasks, even for small degrees of data skew. Consequently, Dedoop supports several automatic load balancing techniques that employ an additional data analysis job to acquire information about data distribution (see Section 3.1).

Classifier Training (optional): For machine learning-based match classification, Dedoop schedules a MapReduce job that trains a classifier based on a set of labeled examples before the actual workflow (the similarity values serve as feature for classification). Dedoop employs the learners of the popular *WEKA* library (e.g., SVM or decision tree) and ships the resulting classifiers to all nodes using Hadoop's distributed cache mechanism.

The user may also include additional MR jobs before and after Dedoop's generated MR jobs, e.g., to determine term weights for similarity measures such as TF/IDF, or to compute the match quality w.r.t. a perfect match result. Preprocessing results are made available for all nodes using the distributed cache.

2.1 User Interface

Dedoop provides a versatile web interface (see Figure 2 for a screenshot) that allows a comfortable ER workflow definition and submission as well as the inspection of computed match results. It supports simultaneous handling of multiple workflows. Each workflow can connect to a different local or remote Hadoop cluster (Figure 2.a)., e.g., on Amazon's EC2. Dedoop greatly simplifies the recurring and laborious task of launching and connecting to a Hadoop cluster on EC2 by providing a rich interface for configuring all required parameters for both EC2 (e.g., region, instance type) and Hadoop (e.g., number of tasks). To simplify data exchange, Dedoop includes a HDFS file manager (Figure 2.b) supporting common file operations (e.g., upload, download of compressed files/directories, and delete). We are currently working on integrating an S3 client in the file manager. The file manager is also linked to a CSV file viewer (Figure 2.c, collapsed) for convenient inspection of input and output files of MapReduce programs. The CSV viewer also facilitates the workflow definition, e.g., to identify and name usable attributes for blocking and similarity computation.

The main part of the UI (Figure 2.d) defines the workflow and has three sections. The Input Data section allows the selection of the input data sources, id attributes, output directory, and (in case of two sources) the specification of corresponding attributes to match with each other. In the Blocking section, we can specify the blocking strategy along with its blocking key generation function(s). The last section, Matching, defines the matchers (attributes and similarity metrics) and match classification. For match classification, the user can choose between a learning-based (shown in the screenshot) and a threshold-based classification. The former requires the specification of a training data set, the *WEKA* classifier and optional classifier-specific parameters. The latter combines the similarity values based on user-specified weights and considers all entity pairs above a threshold as match.

2.2 Implementation

Dedoop is entirely implemented in Java based on *Google's Web toolkit* and the *SmartClient* Ajax RIA System. Client-side actions are processed by Java Servlets on the server side that communicate with the Namenode and the Jobtracker of Hadoop clusters. Dedoop's architecture is designed to easily include new match components. It is built upon an ER component library that comprises all workflow patterns (e.g., data analysis) and custom ER components (e.g., similarity measures or functions for blocking key generation) bundled in a single jar archive, containing executable byte code. Dedoop scans the component library to automatically identify all available executable components. Furthermore, it extracts all relevant parameters (encoded via Java annotations) to be specified in the GUI.

After workflow specification, Dedoop maps the specified workflow to relevant execution units in the component library. Then, for each job driver class a *JobConf*[1] is generated. All *JobConf*s are enriched with typical MR parameters (e.g., input/output directory, #map/reduce tasks) and the corresponding entries for utilized ER components (e.g., similarity metrics). Finally, Dedoop submits the component library along with the generated *JobConf*s to the Hadoop cluster. Dedoop is designed to serve multiple users that may execute multiple workflows simultaneously on the same or on different clusters. To this end, workflow executers (one for each connected cluster) asynchronously consume the submitted workflows from a queue of pending workflows. All clients periodically poll the workflow executers for the progress of their user's workflows and update the GUI.

3. EFFICIENT CLUSTER UTILIZATION

The MR model poses significant challenges for efficient cluster utilization for ER workflows. First blocking-based similarity computation is vulnerable to data skew, i.e., skewed block sizes may lead to severe load imbalances in the reduce phase. Large blocks may therefore 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. Second, in the presence of multiple blocking keys per entity, e.g., when applying multi-pass blocking, the same entity pair may be considered several times, e.g., when two entities share more than one key. This may lead to unnecessary computations, e.g., if the pairs are processed on different nodes. Dedoop therefore provides strategies for load balancing (Section 3.1) and redundant-free pair comparisons (Section 3.2).

3.1 Load Balancing

Dedoop's load balancing approaches are based on an additional MR job that is executed right before the actual matching job. Figure 3 illustrates the general MR workflow. The first MR job analyzes the input data and is the same for all load balancing schemes. The output of this analysis job is a so-called block distribution matrix (BDM) that holds the number of blocking keys separated by input partitions. The BDM not only represents the underlying data skew but also allows for an easy enumeration of blocks, entities of a block, and even of entity pairs to compare. It is then used by a subsequently executed MR job that realizes the load balancing in the map phase and compares entities

[1]Hadoop's interface to describe a map-reduce job.

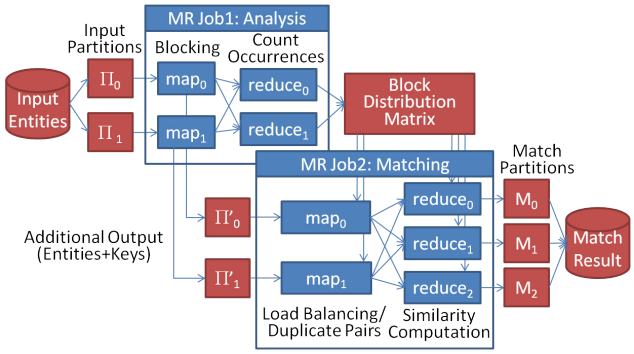

Figure 3: General scheme of Dedoop's efficient cluster utilization.

in the reduce phase. Thereby, the individual tasks need to hold only a small fraction of BDM in memory that depends on the chosen load balancing strategy.

In the following, we sketch our BlockSplit strategy that can be applied to realize load balancing for Standard Blocking. See [4] and [3] for other load balancing strategies, for Sorted Neighborhood, and Standard Blocking, respectively.

BlockSplit generates one or several so-called match tasks per block and distributes match tasks among reduce tasks. It processes small blocks within a single match task. Large blocks are split according to the m input partitions into m sub-blocks. The resulting sub-blocks are then processed using match tasks of two types. Each sub-block is (like any unsplit block) processed by a single match task. Furthermore, pairs of sub-blocks are processed by match tasks that evaluate the Cartesian product of two sub-blocks. This ensures that all comparisons of the original block will be computed in the reduce phase. 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 to make it unlikely that they dominate or increase the overall execution time.

The realization of BlockSplit makes use of the BDM as well as of carefully constructed composite map output keys and corresponding partitioning, sorting, and grouping functions.

3.2 Redundant-free Comparisons

Using a single blocking key may not sufficiently allow finding all duplicates especially with dirty input data. Match workflows therefore often employ multiple blocking keys per entity that are derived from different attribute groups to improve the effectiveness compared to the use of a single blocking key. Unfortunately, multiple blocking keys often lead to overlapping blocks, i.e., two (matching) entities are likely to share two or more keys. A distributed block-wise computation may consequently lead to unnecessary computations if two entities belong to the same two blocks.

Dedoop therefore compares entity pairs only for their smallest common blocking key. To this end, the map output (blocking_key, entity) is annotated with a set (SB) that contains all entity's blocking keys that are smaller than the emitted blocking key. The reduce phase then compares each candidate pair (i.e., entities sharing the same blocking key) if and only if the two SB sets are disjoint. This approach ensures that each candidate pair is compared exactly once.

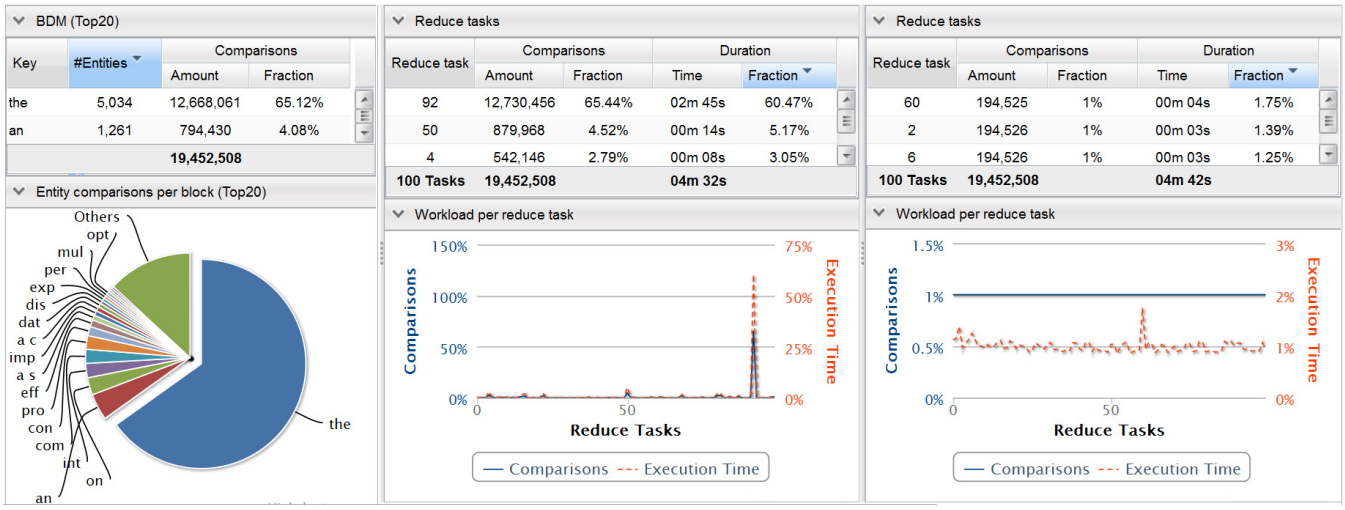

Figure 4: Dedoop's analysis screen visualizes data skew (left), the workload of reduce tasks w/o (middle), and w/ (right) load balancing.

Consider for example two entities A and B with blocking keys x, y, z and w, y, z, respectively. The standard approach would compare both entities twice since they share two blocks (y and z). For block y the SB sets are disjoint ($SB(A_y) = \{x\}$ and $SB(B_y) = \{w\}$) and the entity pair (A, B) is compared. On the other hand, the SB sets for block z are overlapping ($SB(A_z) = \{x, y\}$ and $SB(B_z) = \{w, y\}$) and, thus, A and B are *not* compared.

4. DEMONSTRATION DESCRIPTION

During the demonstration we will illustrate how Dedoop can be employed for entity resolution. In a first scenario we showcase how users can specify and execute ER workflows. To this end we employ our local Hadoop cluster at the University of Leipzig (5 nodes). The second scenario deals with Dedoop's facilities for efficient cluster utilization. We will demonstrate the effect of different load balancing strategies on a large Amazon EC2 cluster with 50 nodes.

Scenario "Data Quality": We will show how Dedoop is used to model and evaluate ER workflow for different datasets of varying size. We demonstrate how to interactively construct an ER workflow approach by selecting and adding relevant components such as blocking keys and similarity measures for attribute matching. Dedoop's UI will automatically ask for the relevant parameter values (see Figure 2). The workflow can then be executed and the match result can be evaluated (precision, recall, and F-measure) w.r.t. a perfect match result. Dedoop's efficient cloud-based execution of ER workflows computes match results very quickly. The audience can, thus, immediately try to improve the match result by adjusting the workflow. For example, recall can be improved by adding another blocking key. Furthermore, a machine learning approach may be selected to find an automatic selection and combination of similarity measures.

Scenario "Efficiency": To demonstrate the effectiveness our load balancing strategies, we provide pre-configured workflows on a large dataset. We first illustrate the data skew, i.e., the number of entities and the resulting number of pair comparisons for all blocks (see Figure 4 left). Dedoop's graphical and tabular views allow for a quick identification large blocks and the degree of data skew. We then execute the workflow without load balancing and thereby illustrate that data skew translates into workload skew. Dedoop's analysis screen visualizes the execution times of all reduce tasks along with their assigned workload (i.e., number of pair comparisons) and proves that the majority of cluster nodes are underutilized (Figure 4 middle). We then re-run the same workflow with the same data but employ one of Dedoop's load balancing strategies. The audience can then compare the overall execution times of both workflows. Dedoop's detailed analysis screen will reveal that all reduce tasks have approx. the same execution time when load balancing has been applied (Figure 4 right).

The audience may also investigate the details and differences between Dedoop's load balancing strategies by manipulating certain parameters, e.g., the number of nodes and the degree of data skew. The audience will therefore be able to assess the robustness as well as the scalability of Dedoop's load balancing strategies. Furthermore, we will provide workflows with multiple blocking key generators to show the influence of eliminating redundant comparisons.

5. REFERENCES

[1] Dedoop. `http://dbs.uni-leipzig.de/dedoop`.

[2] Baxter, Christen, and Churches. A Comparison of Fast Blocking Methods for Record Linkage. In *Workshop Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, 2003.

[3] Kolb, Thor, and Rahm. Load Balancing for MapReduce-based Entity Resolution. In *ICDE*, pages 618–629, 2012.

[4] Kolb, Thor, and Rahm. Multi-pass Sorted Neighborhood Blocking with MapReduce. *CSRD*, 27(1):45–63, 2012.

[5] Köpcke and Rahm. Frameworks for Entity Matching: A Comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.

[6] Köpcke, Thor, and Rahm. Evaluation of Entity Resolution Approaches on real-world Match Problems. *PVLDB*, 3(1):484–493, 2010.