DduP – Towards a Deduplication Framework utilising Apache Spark

Niklas Wilcke

Datenbanken und Informationssysteme (ISYS)
University of Hamburg
Vogt-Koelln-Strasse 30
22527 Hamburg
1wilcke@informatik.uni-hamburg.de

Abstract: This paper is about a new framework called DeduPlication (DduP). DduP aims to solve large scale deduplication problems on arbitrary data tuples. DduP tries to bridge the gap between big data, high performance and duplicate detection. At the moment a first prototype exists but the overall project status is work in progress. DduP utilises the promising successor of Apache Hadoop MapReduce [Had14], the Apache Spark Framework [ZCF$^+$10] and its modules MLlib [MLl14] and GraphX [XCD$^+$14]. The three main goals of this project are creating a prototype of the mentioned framework DduP, analysing the deduplication process about scalability and performance and evaluate the behaviour of different small cluster configurations.

Tags: Duplicate Detection, Deduplication, Record Linkage, Machine Learning, Big Data, Apache Spark, MLlib, Scala, Hadoop, In-Memory

1 Introduction

Duplicate detection is the problem to find all duplicates within a set of elements. Duplicate detection is a performance critical task because of its quadratic complexity. Therefore it seems to be a promising approach to scale horizontally by solving deduplication problems on a distributed environment. On the other hand there are some drawbacks when switching to a classic map reduce framework like Apache MapReduce. We want to avoid these drawbacks by utilising the Apache Spark framework.

The main contributions of this paper are as follows: First, we describe the problems inherent to deduplication frameworks based on Apache MapReduce. Second, we describe the Apache Spark framework and its benefits for deduplication, in particular. Third, we describe architectural concepts of our deduplication framework and detail how it will make use of existing libraries.

2 Related Work

The deduplication process described by Peter Christen in the book "Data Matching" [Chr12b] is the basis for this work. It consists of five steps which shall be briefly explained. The first step is the preprocessing bringing the data into a standardised form. To reduce the quadratic complexity it is followed by a search space reduction, which selects candidate pairs from the search space in an efficient way. Search space reduction is often done by blocking [Chr12a]. The members of each selected pair are compared by a previously chosen set of measures (one per feature) to determine their similarity. Having assigned such a similarity vector to each pair a classifier splits the set into duplicate pairs and no-duplicate pairs. Typically the classification step is followed by a clustering phase [Chr12b]. The last step is the evaluation of the result.

Kolb, Thor and Ram at the University of Leipzig published "Dedoop: Efficient Deduplication with Hadoop" [KTR12]. Dedoop is a Hadoop MapReduce based application to perform deduplication. It is not open source and still a prototype therefore it's hard to investigate or contribute to. Using Hadoop MapReduce as a platform for duplicate detection has the following drawbacks. Hadoop MapReduce is not designed for an interactive usage. It is made for big data batch processing. Especially in the elaborative usage of such a framework an interactive quick responding shell is an advantage. A shell would ease the way exploring the framework and developing own applications.

Optimisation is a huge topic in duplicate detection. For each step in the process described by Christen there are several different algorithms and each one can be parameterised with multiple parameters. Finding the right configuration for a particular data source is a complex problem. Hadoop MapReduce lacks the ability to cache intermediate results to use them for multiple alternatives.

Implementing complex algorithms with Hadoop MapReduce is also difficult because of its low level API. Having high level primitives for distributed computing would simplify development. Nevertheless Hadoop MapReduce is a mature standard for distributed batch processing.

Apache Pig is a high level API for Apache Hadoop. Scripts are written in a procedural language called Pig Latin. Apache Pig enables the user also to store intermediate results in-memory. There is also Apache Hive, which is a data warehousing API on top of Apache Hadoop. Apache Hive provides data access in a declarative way via HiveQL an SQL like querying language. Both high level APIs are not designed to integrate in an existing programming language. Therefor they are not optimised to build huge projects, because the code written in Pig Latin or HiveQL is very hard to refactor.

Apache Spark introduces the new concept of a Resilient Distributed Dataset (RDD). It creates a high level abstraction for distributed fault tolerant in-memory computation. One huge advantage of RDDs is to be storable in memory for later reuse. Especially iterative algorithms benefit from this fact but also a process pipeline gets accelerated. For that reason Spark doesn't depend on slow hard disk I/O assuming a proper cluster configuration. RDDs are mostly distribution transparent. As a result the RDD API feels very much like programming in a non distributed environment. There is no need to learn a new programming language because Spark offers bindings for Scala[1], Java and Python. Apache Spark claims to be the first interactive map reduce like platform. Coming with an interactive shell and a local mode Spark enables the user to start experimenting very quick. Testing and executing Apache Spark applications during development is also very easy. There is no need for a running Spark cluster. Only the spark libraries needs to be present in the project class path and a local cluster will be started on application startup. Other big projects like Apache Mahout [Mah14] are moving from Hadoop MapReduce to Spark. That will enrich the set of available algorithms for machine learning. The Mahout project is also about to build a R[2] like Scala Domain Specific Language (DSL) for Linear Algebra.

Apache Storm and Apache Flink are also two map reduce like computing frameworks in the Apache universe. Apache Flink is a very young project that combines a map reduce like API for distributed computations with query optimisation known from parallel databases. It has also an interactive shell but is still an incubator project and therefore not that mature. Apache Storm is a distributed real time data processing system. Both of them do not offer that much library support like Spark does.

Apache Spark seems to be the most promising candidate for implementing DduP on top. With its lean high level functional API, in-memory approach and programming language integration it meets our demands. Further more it ships with an interactive shell and is very easy to get started with. It is under active development and cutting edge approach in map reduce like distributed computing. There is an ecosystem of several libraries growing up and other projects like Mahout are migrating to Spark.

3 Duplicate Detection Process utilising Apache Spark

We interpret duplicate detection in the most general way. The problem is to find all duplicates within a Set C of tuples called corpus. A tuple is a datum with n features separated by a given scheme. It's mathematical representation is a n-dimensional vector. Each dimension corresponds to a feature. One possible four-dimensional tuple scheme and an example instance could be the following one describing persons.

Person Scheme and Example Instance

```
(forname, lastname, birthdate, place of birth)

(Max, Mustermann, 01.01.1970, Musterstadt)
```

Our duplicate detection process only slightly differs from Christen's previously mentioned one. All tuples get parsed from its input sources and afterwards they get preprocessed.

[1]Scala is a multi paradigm programming language combining object orientation and functional aspects which is executable on a Java Virtual Machine.

[2]R is a programming language focusing on statistics

There are different generic preprocessing steps available. For instance there are trim, to lower and regular expression removal steps. The only allowed type for features is String at the moment. Also numeric features are represented by strings. Strings were chosen because common similarity measures are defined to operate on them. To find typos and other errors introduced by human interaction a string representation seems to be more sensible. Nevertheless other types would also be possible to integrate. Result of the preprocessing is a set of all tuples called corpus.

To reduce the quadratic search space a blocking step filters out possible duplicate pairs in subquadratic time. There are two algorithms implemented. Suffix Array Blocking is implemented by using the Spark core API. Sorted Neighbourhood Blocking is implemented using the SlidingRDD from MLlib. Implementing other sliding window algorithms is very easy utilising SlidingRDD. Blocking is the crucial step in the whole process. Achieving a reduction ratio which is too low results in a long runtime and high memory consumption. A higher reduction ratio corresponds in general to a lower recall and therefore a loss of duplicate pairs. This trade-off is an optimisation problem which at the moment needs to be solved manually. Result of the blocking are tuple candidate pairs.

Every resulting candidate pair is compared featurewise by a similarity measure. There are several different measures available and most of them are normalised to the interval $[0, 1]$. 0 stands for not equal and 1 represents equality. Whether a normalisation is needed depends on the classification algorithm. For every feature dimension a measure needs to be defined. Therefore it's possible to use different measures for different dimensions. All the different measures are imported from the Scala library stringmetric [Str14]. Result of this comparison called similarity step is a similarity vector attached to the each candidate pair.

The binary classification in duplicate and no-duplicate pairs is done by a previously trained decision model. MLlib provides a Decision Tree, a Naive Bayse classifier and a Support Vector Machine (SVM) with a linear kernel for decision modeling. Result of the classification are the two sets duplicates and no-duplicates. The duplicate set containing all observed duplicate pairs is converted into a graph using the Spark library GraphX. Two connected nodes in the graph are representing a duplicate tuple pair. To resolve inconsistencies regarding transitivity a clustering is applied on the graph. The clustering tries to resolve constellations like A is a duplicate of B and B is a duplicate of C but A is not a duplicate of C. The whole process and its intermediate data types are depicted in Figure 1.

In this section we want to discuss briefly how the newly implemented steps scale in a distributed environment. The parsing can be done completely in parallel. Every line respectively tuple representation can be parsed independently. This is also valid for the preprocessing step.

Blocking in general always needs to shuffle the data. It depends on the algorithm which kind of shuffle is needed. Sorted Neighbourhood [Chr12b] for instance needs to sort the whole list of tuples by a key derived from each tuple. As shown in "Minimal MapReduce Algorithms" [TLX13] this can be done by TeraSort in $O(n \cdot log(n))$. Having n nodes in the cluster the fraction of $\frac{n-1}{n}$ of the input tuples needs to be shuffled to different nodes in the average. $\frac{n-1}{n} \approx n$ is valid for large n. That means the whole input set has to be

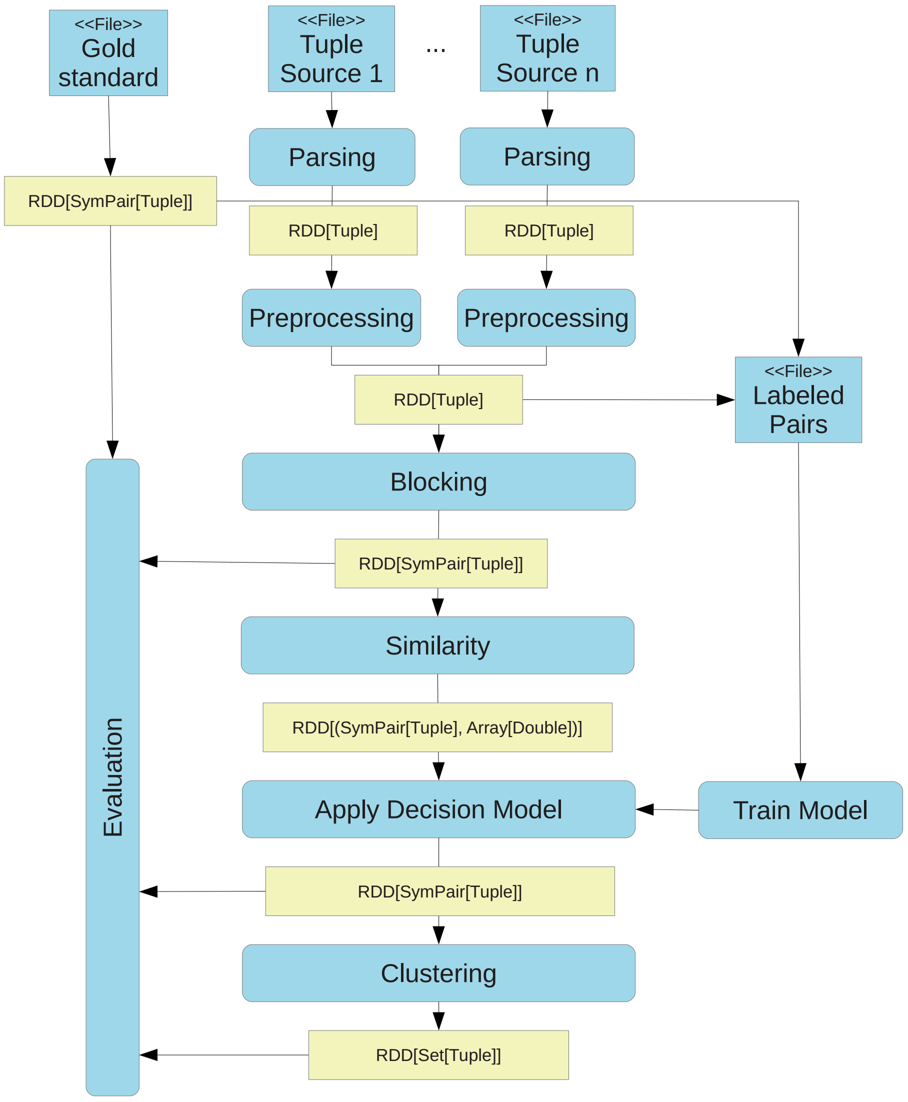

Figure 1: Overview of the separated steps (blue) of the duplicate detection process. The connecting input and output type is illustrated in yellow. RDD is the spark Resilient Distributed Dataset which can be seen as a distributed generic collection. SymPair is a generic symmetric pair.

relocated. Therefore the network bandwidth becomes an important factor.

Calculating the similarity of a tuple pair can be done in parallel. The classification task is completely covered by the existing Spark machine learning library MLlib. Clustering will be done via Sparks GraphX library. In the first step a graph is created with tuples as vertices and tuple pairs as edges. After deleting weak edges the transitive closure will be computed by GraphX. Every resulting component is a cluster of duplicates. Using a graph library for clustering might be an overhead, but the gain is a convenient API to implement cluster algorithms.

4 DduP Framework - Implementation Guideline

DduP aims to be a modular and open source distributed duplicate detection framework. It is implemented in Scala 2.10 utilising the Apache Spark framework.

Main guideline is minimalism and the usage of common standards, best practices and existing libraries. The project tries to focus on topics not covered yet. These topics are preprocessing, search space reduction, pipelining and evaluation in the context of distributed computing.

Figure 1 depicts the deduplication process which is a pipeline consisting of several steps called pipes. A pipe is a generic type combining an input and output type. A pipeline is a sequence of pipes with corresponding input and output types of the neighbouring steps. Steps of the same type can easily be replaced. Using this pattern it is easy to build modular steps for the different phases of the process. The resulting pipeline can be modified easily. This is especially useful for learning environments. There is a Java Framework "Tinkerpop Pipes" [Pip14] for building directed process pipelines. Due to its dependence on iterable collections it is not suitable to process RDDs. The last Spark 1.2 release included an alpha version of a pipelining framework for MLlib. Drawbacks of that framework are, that only SchemaRDDs are allowed to be used what may be inefficient. A SchemaRDD is like relational database table consisting of columns and rows. For that reason we tend to use our own implementation, which is a type safe directed linear pipeline without any type restrictions. The architectural decision whether to use an alpha standard from the MLlib or to use the own implementation is not made yet. Possible libraries to utilise are the following.

MLlib for decision modelling - included in Spark

MLlib Pipelines for modular pipeline design - alpha version included in Spark since 1.2

GraphX for clustering - included in Spark

Sringmetric for similarity calculation - Open source Scala library [Str14]

Weka (optional) for decision modelling and preprocessing - Open source Java machine learning software [Wek14]

Apache Mahout (optional) for decision modelling and preprocessing - Open source Hadoop machine learning library [Mah14]

Besides its output value every pipe in the pipeline produces an evaluation output. Basically there are three input sources an evaluation of a single pipe is based on. These are the corpus containing all tuples, the gold standard containing all true duplicate pairs and the output of the analysed pipe. Currently implemented evaluation measures are set sizes, relative errors, recall and precision. Evaluation is also a performance critical task because it involves expensive calculations. An evaluation log output example of the actual prototype can be seen in Figure 2.

Implementing a wide range of algorithms is not planned. The focus lies on extensibility. There are two different blocking algorithms implemented so far. These are Sorted Neighbourhood [Chr12b] and Suffix Array Blocking [Chr12b]. For clustering there is a base class with functionality to derive clustering algorithms from. The only available implementation at the moment is a transitive closure clustering.

5 Conclusion and Future work

At the moment there is no open source big data deduplication framework available which is scalable with respect to the amount cluster nodes. The briefly introduced deduplication framework called DduP tries to close this gap. It includes pipelining interfaces and implementations for every step mentioned in the overall process. It offers only a small variety of algorithms but aims to be extensible. First tests of the framework are promising but further development and evaluation still needs to be done.

Next step will be the evaluation of the framework. Evaluation will cover different amounts of data and different cluster settings to evaluate how the process scales. Possible cluster settings are 2, 4 and 8 worker nodes plus one master. The main evaluation criteria is the runtime of the process. Performance measures like recall and precision are not that relevant because algorithm performance is not in the focus.

A difficult task is to provide enough data to saturate the cluster. There are two different ways to get test data. The first one is to generate synthetic data and the second one is to use real world data with real duplicates. The usage of real data is difficult, because the gold standard is typically unknown. Combining the two approaches seems to be promising. That means taking a real set of data and randomly duplicate and falsify some tuples by imitating human behaviour. Introducing typos or mistakes based on phonetic equality are only two possibilities.

A set of 10^6 tuples and a size of about 100 MB seems to be a good amount of input data to saturate a small cluster. Depending on the reduction ratio of the search space reduction step this amount of data can easily lead to a very long runtime. Obtaining such a huge set of clean real data is also difficult. In 2014 there were less than $5 \cdot 10^6$ articles in the English Wikipedia [Wik15]. For our purpose data tuples following a scheme are needed and that are much less than $5 \cdot 10^6$ articles. One possible data source could be the Musicbrainz.org

```
##### READING ###########################################
# Number of Tuples:                               13555 #
#-------------------------------------------------------#
# Runtime Reading:                     00d 00:00:00.293 #
# Runtime Analyser:                    00d 00:00:01.602 #
#########################################################
##### GOLD STANDARD #####################################
# Number of Pairs:                                 5754 #
#-------------------------------------------------------#
# Runtime Gold Standard:               00d 00:00:00.059 #
# Runtime Analyser:                    00d 00:00:02.490 #
#########################################################
##### BLOCKING ##########################################
# Reduced search space size:                      14200 #
# Naive search space size:                     91862235 #
# Reduction ratio:                          0,999845421 #
# Max possible recall:                      0,625825513 #
#-------------------------------------------------------#
# Runtime Blocking:                    00d 00:00:00.055 #
# Runtime Analyser:                    00d 00:00:04.350 #
#########################################################
##### TRAINING SET BUILDING #############################
# Training and test set size:                       200 #
#-------------------------------------------------------#
# Runtime Training Set Building:       00d 00:00:00.439 #
# Runtime Analyser:                    00d 00:00:00.188 #
#########################################################
##### DECISSION TRAINING ################################
# Training set size:                                121 #
# Test set size:                                     79 #
# Training Error:                           0,075949367 #
#-------------------------------------------------------#
# Runtime Decission Training:          00d 00:00:01.160 #
# Runtime Analyser:                    00d 00:00:00.457 #
#########################################################
##### DECISSION #########################################
# Number of duplicate pairs found:                 9809 #
#-------------------------------------------------------#
# Runtime Decission:                   00d 00:00:00.015 #
# Runtime Analyser:                    00d 00:00:00.775 #
#########################################################
##### CLUSTERING ########################################
# Recall:                                   0,551963851 #
# Precission:                               0,341578834 #
#-------------------------------------------------------#
# Number of clusters:                              2865 #
# Average cluster size:                     2,769633508 #
#-------------------------------------------------------#
# Runtime Clustering:                  00d 00:00:04.837 #
# Runtime Analyser:                    00d 00:00:01.993 #
#########################################################
```

Figure 2: Sample log output of the current prototype detecting duplicates in a small example data set. Due to Sparks lazy evaluation the runtimes are not always correctly separated into pipe and analyser runtime.

[Mus14] database which is a free music database. The database has a complex scheme and contains many informations about released music albums. The first application of the DduP framework could be the deduplication of the Musicbrainz database.

Another approach to get input data seems to be sensible. Taking a set of clean real input data for each feature dimension and do a cartesian product over all dimensions. Having n feature dimensions with k different real data values each, the resulting test set size is k^n. That means with 4 feature dimensions and 32 different values our target of 10^6 is reached. This theoretical data set is not usable because there are many tuples only varying in one dimension what would be a duplicate. Increasing the size of dimensions and data values per dimension solves this problem.

For now there is only one very old tool called "DbGen" [DbG14] from the University Texas in Austin which can do this kind of data generation. It produces clusters of falsified person tuples. Falsification and other parameters can be adjusted. Basis for the generation are a set of 62491 different names and 42115 different zip codes. Addresses and social security numbers gets generated. It looks like we have to implement own tools to prepare test data.

References

[Chr12a] Peter Christen. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, 24(9):1537–1555, Sept 2012.

[Chr12b] Peter Christen. Data matching. *Data-Centric Systems and Appl., Springer*, 2012.

[DbG14] DbGen. `http://www.cs.utexas.edu/users/ml/riddle/index.html`, 2014.

[Had14] Apache Hadoop. `http://hadoop.apache.org`, 2014.

[KTR12] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[Mah14] Apache Mahout. `http://mahout.apache.org`, 2014.

[MLl14] Apache Spark MLlib. `https://spark.apache.org/mllib`, 2014.

[Mus14] Musicbrainz. `https://musicbrainz.org/`, 2014.

[Pip14] Tinkerpop Pipes. `http://www.tinkerpop.com`, 2014.

[Str14] Stringmetric. `https://github.com/rockymadden/stringmetric`, 2014.

[TLX13] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *Proceedings of the 2013 international conference on Management of data*, pages 529–540. ACM, 2013.

[Wek14] Weka. `http://www.cs.waikato.ac.nz/ml/weka`, 2014.

[Wik15] Wikipedia. `https://en.wikipedia.org/wiki/Wikipedia:Size\_of\_Wikipedia`, 01 2015.

[XCD$^+$14] Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.