Подход к реализации методов разрешения сущностей в среде распределенных вычислений Hadoop/MapReduce

© М.Д. Ислентьев

Московский государственный университет им. М.В. Ломоносова,
Москва, Россия

mikhail.islentyev@gmail.com

Аннотация. Процесс разрешения сущностей является частью процесса интеграции данных. Разрешение сущностей зависит, кроме всего прочего, от выбора мер сходства значений и методов сравнений пар записей. Данная статья посвящена разработке подхода к реализации мер сходства строковых значений как наиболее распространенных типов данных, а также детерминированных методов сравнения пар записей в среде распределенных вычислений Hadoop/MapReduce с использованием языка высокого уровня Jaql.

Ключевые слова: большие данные, интеграция данных, разрешение сущностей, связывание записей, удаление дубликатов, Hadoop, MapReduce, Jaql.

An Approach for Implementation of Methods for Entity Resolution in the Hadoop/MapReduce Distributed Computing Environment

© Mikhail D. Islentyev

Lomonosov Moscow State University,
Moscow, Russia

mikhail.islentyev@gmail.com

Abstract. The process of entity resolution is part of the data integration process. Entity resolution depends on the choice of similarity measures for values and methods for comparing pairs of records. In this paper, we develop an approach to implementation similarity measures of string values as the most common types of data, as well as deterministic methods for comparing pairs of records in the Hadoop/MapReduce distributed computing environment using the high-level language Jaql.

Keywords: big data, data integration, entity resolution, record linkage, data deduplication, Hadoop, MapReduce, Jaql.

1 Введение

В различных областях науки наблюдается экспоненциальный рост объема получаемых экспериментальных данных. Сложность использования таких данных увеличивается еще и вследствие их естественной разнородности. Это неизбежно приводит к необходимости использования неоднородной, распределенной информации, накопленной в течение значительного периода наблюдений различными инструментами.

Для анализа больших объемов данных используются современные среды распределенных вычислений, такие, как Hadoop/MapReduce [14, 19]. Такие среды имеют почти линейную горизонтальную масштабируемость и высокую отказоустойчивость. Основным достоинством подобных сред является возможность анализировать и обрабатывать разно-структурированные данные (реляционные, JSON, XML, тексты и др.). При этом возникает проблема интеграции данных, извлекаемых из разно-структурированных источников. Традиционно процесс интеграции данных включает в себя следующие шаги: унификация моделей данных, сопоставление схем, разрешение сущностей [6, 16] и слияние данных [5]. Остановимся подробней на этапе разрешения сущностей.

Данный этап нацелен на поиск записей в одном или нескольких наборах данных, представляющих собой один и тот же объект в реальном мире, или сущность. Он ориентирован на решение таких задач, как связывание записей, выявление и удаление дубликатов, сопоставление связей и др.

Труды XIX Международной конференции «Аналитика и управление данными в областях с интенсивным использованием данных» (DAMDID/ RCDL'2017), Москва, Россия, 10–13 октября 2017 года

Весь процесс разрешения сущностей, в соответствии с [10], можно разделить на следующие этапы: подготовка данных; выбор мер сходства значений; выбор метода сравнения пар записей; определение ограничений.

Данная работа сосредоточена на втором и третьем этапах. Выбор мер сходства является одним из самых важных этапов в процессе разрешения сущностей. Важно выбрать меры, наиболее подходящие для представленного набора данных, поскольку именно на основе значений этих мер будет делаться вывод, являются ли записи в паре совпадающими, то есть относящимися к одной сущности, или же нет. И поскольку большинство данных представлено в виде строковых значений (имена, названия, адреса и т. д.), особое внимание уделяется нами мерам сходства строк. Среди методов сравнения пар записей известны детерминированные методы, к которым относятся метод взвешенной суммы и метод, основанный на формулировании правил. Кроме того, отдельно стоят методы разделения на блоки. Они не входят в традиционный процесс разрешения сущностей, однако могут значительно увеличить скорость выполнения и производительность всего процесса при помощи эффективного создания небольших блоков, содержащих только потенциально совпадающие записи.

Нашей целью является разработка подхода к реализации методов разрешения сущностей в среде распределенных вычислений Hadoop/MapReduce. Для непосредственной реализации мер и методов в среде Hadoop/MapReduce был выбран язык высокого уровня Jaql [4]. Jaql – это функциональный декларативный язык запросов, который предназначен для обработки больших наборов данных. Он позволяет работать с разно-структурированными данными, распределенной файловой системой HDFS и, при необходимости, сам переписывает высокоуровневые запросы в запросы «низкого уровня», состоящие из MapReduce-задач.

На языке Jaql реализован ряд мер сходства строковых значений, алгоритмов сравнения пар записей и разбиения записей на блоки, рассмотрены некоторые идеи и особенности реализации. Выбранный набор мер сходства строк обширен, и в большинстве случаев его достаточно для задач разрешения сущностей [17], однако реализация позволяет определять собственные меры сходства, в том числе и для значений, не являющимися строковыми, в виде непосредственно функций на Jaql или же в виде Java UDF (пользовательских функций Java). Код реализации доступен в git-репозитории [12].

В разделе 2 описаны меры сходства строковых значений, приводятся примеры использования и реализации некоторых мер и примеры задания и использования компаратора для получения вектора сравнения пары записей. В разделе 3 описаны детерминированные методы сравнения пар записей и показаны примеры их использования. В разделе 4 обсуждается тема разбиения на блоки для уменьшения числа попарных сравнений и приведены примеры их применения. Наконец, в разделе 5 показан пример полного процесса разрешения сущностей в рамках реализованных мер и методов.

2 Меры сходства значений

Ключевым моментом в процессе разрешения сущностей является получение оценок сходства значений соответствующих атрибутов двух сравниваемых записей путем вычисления мер их сходства. Полученные оценки формируют вектор сравнения, на основе которого в дальнейшем делаются выводы о совпадении или различии рассматриваемой пары записей. Таким образом, важно выбрать наиболее подходящие под имеющиеся данные меры сходства.

Подавляющее большинство значений атрибутов представляет собой строки, и, кроме того, верное определение сходства строк не всегда является тривиальным действием, поэтому особое внимание уделим мерам сходства именно строковых значений. По своей природе их обычно делятся на следующие группы:

- меры сходства на основе редактирования;
- меры сходства на основе разбиения на токены;
- гибридные меры сходства.

Рассмотрим подробнее каждую группу.

2.1 Меры сходства на основе редактирования

Меры данного типа оперируют числом вставок, удалений, замен и/или перестановок символов в строке. Чем больше требуется таких операций для преобразования одной строки к другой, тем менее они похожи.

Для реализации мер этой группы в основном применяется метод динамического программирования [11, 18]. Декларативные языки программирования, к которым относится Jaql, не предназначены для реализации подобных методов, поэтому было решено воспользоваться расширяемостью языка при помощи Java UDF.

Каждая такая функция принимает два строковых параметра (см. Программу 1). В данном примере показан вызов функции расстояния Джаро–Винклера [7] для строк “Dwayne” и “Duane”.

Программа 1 Вызов функции меры сходства на основе редактирования на примере расстояния Джаро–Винклера

```
import similarity;
similarity::jaroWinklerSim("Dwayne",
"Duane");
```

Реализованные меры на основе редактирования: расстояние Левенштейна [7], расстояние Дамерау–Левенштейна [7], наибольшая общая подпоследовательность [11], расстояние Джаро [7], расстояние Джаро–Винклера.

2.2 Меры сходства на основе разбиения на токены

Принадлежащие к этой группе меры используют разбиение строк на токены. Чаще всего применяют

два вида разбиения: разбиение на слова и разбиение на n-граммы.

Для расчета меры сходства набор токенов представляют либо в качестве множества, и тогда применяют меры сходства множеств, либо в качестве вектора в многомерном пространстве, после чего используют меры сходства векторов.

Реализация данных мер подразумевает нахождение пересечения двух наборов токенов, что просто и эффективно реализуется на Jaql с помощью встроенной функции join (см. Программу 2). В качестве примера приведем реализацию коэффициента Дайса [7]:

$$sim_{Dice}(x, y) = \frac{2n_t}{n_x + n_y},$$

где x, y – сравниваемые строки, n_x, n_y – число токенов в строке x и y соответственно, n_t – число совпадающих токенов в строках x и y.

Программа 2 Реализации функции меры сходства на основе разбиения на токены на примере коэффициента Дайса

```
bagsIntersection = fn(lhs, rhs)
  join lhs, rhs
  where lhs.token == rhs.token
  into {
    lhs.token,
    count: min([ lhs.count, rhs.count ])
  };

diceSim = fn(lhs, rhs)
  count(bagsIntersection(lhs, rhs)) *
  2.0 / (count(lhs) + count(rhs));
```

Функции мер данной группы принимают строки, предварительно разбитые на токены.

Список реализованных мер сходства на основе разбиения на токены: коэффициент Дайса, коэффициент Жаккара [7], коэффициент перекрытия [7], косинусный коэффициент [7], статистическая мера TFIDF [7].

2.3 Гибридные меры сходства

К таким мерам сходства относятся меры, оперирующие наборами токенов и применяющие к сравнению токенов меры на основе редактирования. Такие меры отличают повышенная точность оценивания сходства, но в то же время увеличенное время выполнения [7]. Поскольку названные меры также используют наборы токенов, реализация этих методов тоже была выполнена на языке Jaql. Были реализованы следующие гибридные меры сходства: сходство Монг-Элкана [7] и статистическая мера SoftTFIDF [7].

2.4 Вектор сравнения

Вектор сравнения, как было сказано ранее, формируется из оценок сходства для значений атрибутов пар записей. В реализации используется компаратор в виде записи, атрибутами которой являются имена оценок (используются далее в методах сравнения пар записей), а значениями – функции мер сходства (см. Программу 3).

Программа 3 Пример задания компаратора

```
import similarity;
addressComparator = fn(lhs, rhs)
  similarity::jaccardSim(
    lhs.address -> similarity::nGramBag(),
    rhs.address -> similarity::nGramBag()
  );
nameComparator = fn(lhs, rhs)
  similarity::minLengthMongeElkanSim(
    lhs.name -> similarity::wordBag(),
    rhs.name -> similarity::wordBag()
  );
typeComparator = fn(lhs, rhs)
  similarity::equalSim(lhs.type, rhs.type);

recordComparator = {
  addressScore: addressComparator,
  nameScore: nameComparator,
  typeScore: typeComparator
};
```

Далее этот компаратор может быть передан функции вместе с парой записей, для которых необходимо вычислить вектор сравнения (см. Программу 4).

Программа 4 Пример получения вектора сравнения

```
import resolution;
pair = [ { ... }, { ... } ];
vector = recordComparator
  -> resolution::countVector(pair);
```

Реализация функции countVector() получает от компаратора пары атрибут/значение и создает вектор сравнения – запись с теми же именами атрибутов, значения которых являются оценками сходства (см. Программу 5).

Программа 5 Реализация функции countVector()

```
countVector = fn(recordComparator, pair)
  recordComparator -> fields()
    -> transform {
      ($[0]): _evalFieldComparator(
        $[1], pair[0], pair[1]
      )
    } -> record();
```

Закрытая функция _evalFieldComparator() выполнена в виде Java UDF и просто вызывает переданную функцию меры для пары записей. Такой подход к реализации позволил сохранить возможность выполнения данной функции внутри MapReduce-задачи, поскольку явный вызов функции через индексатор ([1]) не позволяет движку оптимизации Jaql создать MapReduce-задачу.

3 Детерминированные методы сравнения пар записей

Пусть имеются пара записей и их вектор сравнения, полученный при помощи компаратора, объявленного ранее:

```
vector = { addressScore, nameScore, typeScore
}
```

Возникает проблема, как по вектору сравнений определить, являются ли записи совпадающими или нет.

3.1 Взвешенная сумма

Наиболее простым решением этой проблемы

является использование среднего значения всех оценок сходства из вектора сравнений или же их взвешенной суммы, после чего необходимо задать пороговое значение, определяющее совпадение или различие записей, например (веса и порог здесь и далее выбраны случайно):

```
0.2 * addressScore + 0.8 * nameScore > 0.85
```

В реализации данного метода вектор весов также представляет собой запись с теми же именами атрибутов, что и у компаратора, значения которого и являются весами (см. Программу 6).

Программа 6 Пример метода взвешенной суммы

```
import
  resolution::classifier::weightedSum as ws;
vector -> ws::classifier({
    addressScore: 0.2,
    nameScore: 0.8
  },
  0.85
);
```

3.2 Правила

Несколько иным способом является формулирование набора правил, где ограничения накладываются на оценки сходства каждого атрибута независимо. Пример таких правил (оператор ‘&’ обозначает логическое И, оператор ‘|’ — логическое ИЛИ):

```
typeScore = 1.0 &
(nameScore > 0.7 | addressScore > 0.9) |
nameScore > 0.9
```

Метод сравнения, использующий данные правила, считает записи в паре совпадающими, если поле `nameScore` их вектора сравнений больше 0.9, или же если `typeScore` равно 1.0, и при этом либо `nameScore` больше 0.7, либо `addressScore` больше 0.9.

Правила реализуются записью, представляющей собой двоичное дерево выражений. В следующем примере составлено дерево, эквивалентное описанным выше правилам (см. Программу 7).

Программа 7 Пример метода на основе правил

```
import resolution::classifier::ruleBased as
rb;
rules = {
  "lhs": {
    "lhs": {
      "lhs": "typeScore",
      "op": "=",
      "rhs": 1.0
    },
    "op": "&",
    "rhs": {
      "lhs": {
        "lhs": "nameScore",
        "op": ">",
        "rhs": 0.7
      },
      "op": "|",
      "rhs": {
        "lhs": "addressScore",
        "op": ">",
        "rhs": 0.9
      }}},
  "op": "|",
  "rhs": {
    "lhs": "nameScore",
    "op": ">",
    "rhs": 0.9
  }};
vector -> rb::classifier(rules);
```

Даже такое небольшое количество правил требует построения достаточно громоздкого двоичного дерева выражений. Для облегчения формулирования и применения правил был реализован синтаксический анализатор в виде Java UDF, позволяющий получить дерево выражений из строкового выражения (см. Программу 8).

Программа 8 Пример разбора строкового выражения правил

```
import resolution::classifier::ruleBased as
rb;
rb::parseRules(
  "typeScore = 1.0 & " +
  "(nameScore > 0.7 | addressScore > 0.9) |
" +
  "nameScore > 0.9"
);
```

Результатом данного вызова функции будет дерево, эквивалентное вышеописанному.

4 Методы разделения на блоки

С увеличением числа данных полное попарное сравнение становится крайне неэффективным. Действительно, полное попарное сравнение набора данных, состоящих из n записей, потребует $n(n-1)/2$, или $O(n^2)$, сравнений. Для уменьшения числа пар записей, которые необходимо сравнить, используются методы разделения на блоки. Такие методы отвергают заведомо несовпадающие пары записей и создают блоки, состоящие из пар, которые потенциально могут совпадать.

Для рассмотрения были выделены следующие методы: метод исключительного разделения на блоки [13], метод индексации биграмм [2] и метод на основе кластеризации с помощью canopy [15]. Реализация этих методов разделена на две части: одна – для задачи выявления и удаления дубликатов (в одном наборе данных), вторая – для задачи связывания записей (для двух наборов данных).

4.1 Метод исключительного разделения

Самым простым является метод исключительного разделения на блоки. Идея метода состоит в том, что набор данных делится на непересекающиеся блоки по блочному ключу. Такой ключ может являться значением какого-либо атрибута записи или же комбинацией нескольких значений или даже их частью.

Реализация данного метода подразумевает группировку записей по блочному ключу, что в языке Jaql возможно совершить с помощью встроенной функции group by. Применение такого метода требует задания функции, генерирующей блочный ключ (см. Программу 9).

Программа 9 Пример метода исключительного разделения для двух наборов данных

```
import
resolution::linkage::blocking::simple;
```

```
data1 = read(...);
data2 = read(...);
data1Key = fn(value)
  value.lastname -> substring(0, 4);
data2Key = fn(value)
  value.surname -> substring(0, 4);

simple::blocking(data1, data2,
  data1Key, data2Key
);
```

Преимуществом этого метода является высокая скорость его исполнения. К недостаткам же можно отнести сложность выбора критерия, а также то, что при не совсем оптимальном его выборе реально совпадающие записи могут попасть в различные блоки, тем самым они никогда не будут сравниваться.

4.2 Индексация биграмм

Следующий метод, называемый методом индексации биграмм, позволяет осуществлять приближенное разделение на блоки. Его алгоритм описан в [2]. Реализация метода сопряжена с получением инвертированного индекса и группировкой, с чем Jaql прекрасно справляется, используя встроенные функции group by и expand unroll. Для проведения такого разбиения необходимо помимо функции блочного ключа подать пороговое значение (см. Программу 11).

Программа 11 Пример метода индексации биграмм для одного набора данных

```
import

resolution::deduplication::blocking::bigram;
data = read(...);
dataKey = fn(value)
  value.address
    -> strSplit("\\s+") -> index(0);
data -> bigram::blocking(dataKey, 0.3);
```

4.3 Кластеризация canopy

Иной подход к разделению на блоки реализует метод, основанный на кластеризации с помощью canopy. Данный метод кластеризации опирается на возможность для случайной записи из набора данных эффективно найти все близлежащие записи при помощи какой-либо приближенной функции расстояния, требующей небольших вычислительных затрат. После проведения кластеризации каждый canopy-кластер формирует свой блок.

Данный алгоритм кластеризации имеет вариант реализации непосредственно на MapReduce, поэтому реализован он был с помощью явного задания MapReduce-задачи на языке Jaql. Для применения метода разделения, основанного на такой кластеризации, необходимо задать функцию расстояния и два пороговых значения (см. Программу 12).

Программа 12 Пример метода кластеризации canopy для одного набора данных

```
import similarity;
import

resolution::deduplication::blocking::canopy;
data = read(...);
distanceFunction = fn(lhs, rhs)
  1.0 - similarity::minLengthMongeElkanSim(
    lhs.name -> similarity::wordBag(),
    rhs.name -> similarity::wordBag()
  );
data -> canopy::blocking(
  distanceFunction, 0.08, 0.16
);
```

Метод индексации биграмм и метод на основе кластеризации с помощью canopy генерируют пересекающиеся блоки, что снижает вероятность разделения на разные блоки совпадающих записей. Также эти методы имеют хороший и близкий друг к другу результат по качеству разделения при правильном выборе функций и порогов, что отражено в [2].

4 Пример использования

Для примера рассмотрим применение процесса разрешения сущностей для одного набора данных (см. Программу 13).

Программа 13 Пример процесса разрешения сущностей для одного набора данных

```
import similarity;
import
  resolution::classifiers::weightedSum    as
ws;
import resolution::deduplication;
import

resolution::deduplication::blocking::canopy;

data = read(...);
distanceFunction = fn(lhs, rhs)
  1.0 - similarity::minLengthMongeElkanSim(
    lhs.name -> similarity::wordBag(),
    rhs.name -> similarity::wordBag()
  );
blockingInfo = canopy::createInfo(
  distanceFunction, 0.08, 0.16
);
addressComparator = fn(lhs, rhs)
  similarity::jaccardSim(
    lhs.address -> similarity::nGramBag(),
    rhs.address -> similarity::nGramBag()
  );
nameComparator = fn(lhs, rhs)
  similarity::minLengthMongeElkanSim(
    lhs.name -> similarity::wordBag(),
    rhs.name -> similarity::wordBag()
  );
typeComparator = fn(lhs, rhs)
  similarity::equalSim(lhs.type, rhs.type);

recordComparator = {
  addressScore: addressComparator,
  nameScore: nameComparator,
  typeScore: typeComparator
};

classifierInfo = ws::createInfo({
    addressScore: 0.2,
    nameScore: 0.8
  },
  0.85
);
data                                        ->
deduplication::deduplicateWithBlocking(
  blockingInfo,
  recordComparator,
  classifierInfo
);
```

Данные после чтения будут разделены на блоки с помощью метода кластеризации canopy, затем блоки преобразуются в пары записей, для каждой такой пары будет вычислен их вектор сравнения на основе трех переданных функций мер, по этому вектору метод взвешенной суммы определит, являются ли записи в паре совпадающими или нет, после чего будут возвращены только те пары, записи в которых были определены как совпадающие.

5 Заключение и дальнейшая работа

Мы описали меры сходства строковых значений, детерминированные методы сравнения пар записей и разработали подход к их реализации в среде распределенных вычислений Hadoop/MapReduce с использованием высокоуровневого языка программирования Jaql. Также описаны и реализованы методы по разделению пар записей на блоки для значительного снижения числа попарных сравнений и увеличения производительности.

Набор реализованных функций мер сходства строковых значений достаточно обширен, однако реализация позволяет определять собственные меры сходства, в том числе и для значений, не являющимися строковыми, в виде непосредственно функций на Jaql или же в виде Java UDF.

В дальнейшем планируется реализовать поддержку ограничений [10] и метод корреляционной кластеризации [8] для задачи выявления и удаления дубликатов, а также рассмотреть возможность реализации иных методов сравнения пар записей в среде Hadoop, таких, как вероятностные методы [9] и методы, основанные на машинном обучении [1, 3].

Поддержка

Работа выполнена при поддержке РФФИ (гранты 15-29-06045, 16-07-01028).

Литература

[1] Arasu, A., Götz, M., Kaushik, R.: On Active Learning of Record Matching Packages. Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of Data, pp. 783-794. ACM, New York (2010)

[2] Baxter, R., Christen, P., Churches, T.: A Comparison of Fast Blocking Methods for Record Linkage. Proc. of the KDD-03 Workshop on Data Cleaning, Record Linkage and Object Consolidation, pp. 25-27. ACM, New York (2003)

[3] Bellare, K., Iyengar, S., Parameswaran, A.G., Rastogi, V.: Active Sampling for Entity Matching. Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 1131-1139. ACM, New York (2012)

[4] Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin A., Eltabakh, M.Y., Kanne, C.C., Özcan, F., Shekita, E.J.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. Proc. of the VLDB Endowment, 4 (12), pp. 1272-1283 (2011)

[5] Bleiholder, J., Naumann, F.: Data Fusion. ACM Computing Surveys, 41 (1), pp. 1:1–1:41 (2009)

[6] Christen, P.: Data Matching – Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer, Heidelberg (2012)

[7] Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A Comparison of String Distance Metrics for Name-Matching Tasks. Proc. of the 2003 Int. Conf. on Information Integration on the Web, pp. 73-78. AAAI Press (2003)

[8] Elsner, M., Schudy, W.: Bounding and Comparing Methods for Correlation Clustering Beyond ILP. Proc. of the Workshop on Integer Linear Programming for Natural Language Processing, pp. 19-27. Association for Computational Linguistics, Stroudsburg (2009)

[9] Fellegi, I.; Sunter, A.: A Theory for Record Linkage. J. of the American Statistical Association, 64 (328), pp. 1183–1210 (1969)

[10] Getoor, L., Machanavajjhala, A.: Entity Resolution for Big Data. Proc. of the 19th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, p. 1527. ACM, New York (2013)

[11] Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Communications of the ACM, 18 (6), pp. 341-343 (1975)

[12] Islentyev, M.D.: mislen/jaql-entity-resolution: Entity Resolution Methods on Jaql (2017). https://github.com/mislen/jaql-entity-resolution

[13] Jaro, M.A.: Advances in Record Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. J. of the American Statistical Association, 84 (406), pp. 414-420 (1989)

[14] Maitrey, S., Jha, C.K.: MapReduce: Simplified Data Analysis of Big Data. Procedia Computer Science, 57, pp. 563-571 (2015)

[15] McCallum, A., Nigam, K., Ungar, L.H.: Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 169-178. ACM, New York (2000)

[16] Naumann, F., Herschel, M.: An Introduction to Duplicate Detection. Morgan and Claypool Publishers (2010)

[17] Vovchenko, A.E., Kalinichenko, L.A., Kovalev, D.Y.: Methods of Entity Resolution and Data Fusion in the ETL-Process and their Implementation in the Hadoop Environment. Informatics and Applications, 8 (4), pp. 94-109 (2014)

[18] Wagner, R.A.; Fischer, M.J. The String-to-String Correction Problem. J. of the ACM, 21 (1), pp. 168-173 (1974)

[19] White, T.: Hadoop: The Definitive Guide, 4th Edition. O'Reilly Media (2015)