Optimization of Data Cleaning Programs

Tiago Bartolomeu Luiz

Thesis to obtain the Master of Science Degree in

Computer Engineering

Supervisor: Prof. Helena Isabel De Jesus Galhardas

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. Helena Isabel De Jesus Galhardas
Member of the Committee: Prof. Paolo Romano

November 2020

Resumo

Derivado de um mundo totalmente conectado à internet, são recolhidas grandes quantidades de dados a cada segundo. Contudo, grande parte destes dados estão corrompidos, carecendo de tratamento por parte de uma ferramenta de limpeza de dados. Assim sendo, as ferramentas de limpeza de dados precisam de ter a capacidade de processar grandes quantidades de dados de forma eficaz e rápida. No entanto, manter a performance e eficácia é algo não trivial. Estas ferramentas dependem de algoritmos complexos para realizar as tarefas que permitem limpar os dados. Por exemplo, a implementação naïve da deteção de duplicados aproximados tem uma complexidade quadrática - proibitiva quando há milhões de registos. Nós propomo-nos a implementar um otimizador que irá ser incorporado no CLEENEX, uma ferramenta de investigação de limpeza de dados. Este otimizador irá escolher o algoritmo que melhor se adequa à execução de uma dada operação de dados. Considera-se um algoritmo como o mais adequado quando este nos garante o melhor trade-off possível entre performance e qualidade dos resultados.

Palavras-chave:Limpeza de Dados, Optimização de Queries, Base de Dados Relacionais, Deteção de Duplicados Aproximados, Otimização de Performance

Abstract

As a result of an always-online modern world, large amounts of data are being collected every second. However, some of that data is dirty and needs to be cleaned by a data cleaning tool. Therefore, data cleaning tools need to be able to process large amounts of data with a good performance and effectiveness. Maintaining the performance and effectiveness for large amounts of data is difficult because these tools rely on complex algorithms to perform data cleaning tasks. For example, the naïve implementation of the approximate duplicate detection task has a quadratic complexity - unfeasible when there are millions of records. We propose to implement an optimizer to be incorporated in CLEENEX, a data cleaning research prototype. The optimizer will choose the best-suited algorithms to perform each data operation. The algorithm is selected based on the best trade-off between performance and quality of results.

Keywords: Data Cleaning, Query Optimization, Relational Databases, Approximate Duplicate Detection, Performance Optimization

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the modern always-online world, data about each individual is being collected every second. Companies such as Google and Facebook store in their databases data that gives them access to relevant information, such as the users' interests, location at a given time and day, and more information that meet the companies' interests. However, the capability of extracting interesting and useful information is directly correlated to the quality of the data stored in those databases. Data quality can be affected by errors, missing values, duplicates, and inconsistencies. Moreover, data may not be in a format that is proper for consumption, thus needing some transformations. Data cleaning is the process that aims at purifying raw data, and producing data of good quality.

Although there are several software tools that enable to effectively perform data cleaning (e.g., Trifacta[1], Informatica[2] (commercial tools), CLEENEX [15] and BigDansing [22] (research prototypes), most of them fail at efficiently perform that task when handling large amounts of data. Typically, data cleaning tools rely on complex algorithms to perform data cleaning tasks such as deduplication (i.e., the process of detecting and eliminating approximate duplicates).

Data cleaning tools are typically rule-based or transformation-based. In *rule-based tools*, the user defines a set of rules that data of good quality must satisfy. Each time an entry on a dataset does not satisfy a given rule, it is considered a violation, and needs to be repaired. One or more data repairs are defined by the user for each violation. Data repairs are typically selected from the set of possible data repairs using heuristics. In *transformation-based tools*, we define a graph of transformations that the input data must go through. These transformations are performed by data cleaning operators, that transform dirty data in clean data. Naïve algorithms for deduplication have quadratic complexity, since they perform a Cartesian product to compare every pair of records in a given input dataset. Therefore, Cartesian product should be avoided because it has a significant impact in the deduplication performance. Some techniques have been proposed to avoid the Cartesian product by limiting the comparisons performed. However, these techniques may not be able to correctly identify all true duplicates. Therefore, there is the need to find a good trade-off between performance (efficiency) and the capability of finding the approximate duplicates (effectiveness).

[1] https://www.trifacta.com
[2] https://www.informatica.com

1.1 The CLEENEX Data Cleaning Tool

CLEENEX is a transformation-based data cleaning tool. That is, it represents the transformations that the data goes through in a graph. CLEENEX also allows the user to define a set of rules that the records must satisfy, known as *Quality Constraints* (QCs). For example, a quality constraint could be that the output of a data transformation does not contain null values. Moreover, when a quality constraint is not satisfied, the user can repair the faulty data by applying one or more *Manual Data Repairs* (MDRs).

In CLEENEX there is a clear division between the logical operators, declared through an extension of the SQL language or a Graphical-User Interface (GUI), and the physical operators, that define the algorithms that implement those logical operators. This separation allows us to focus on optimizing physical operators to enhance their performance. This separation resembles the architecture of a Relational Database Management System (RDBMS). Moreover, CLEENEX enables user intervention during the data cleaning process, allowing him, for example, to apply one or more MDRs if a QC is not satisfied, as well as data debugging to analyze the source of potential problems affecting data quality.

The reason why this thesis focus on CLEENEX is because we have access to its code base. This is majorly due to the fact that CLEENEX is a research tool under development at Instituto Superior Técnico, unlike the remaining data cleaning tools we refer to in this thesis.

1.2 Problem

The problem addressed in this thesis consists on optimizing a data cleaning process in CLEENEX. A data cleaning process is represented by a graph of transformations that the user defines. This graph of transformations is composed by two kinds of nodes: *(i)* the data transformation nodes, and *(ii)* the data transformation output nodes. We will work in optimizing the first type of nodes.

CLEENEX does not have an automatic optimizer, thus, it is not able to choose the most efficient graph of transformations to perform a data cleaning process. Moreover, when faced with large amounts of data, CLEENEX may be unable to execute a data cleaning process, especially if it involves expensive processes such as deduplication. This problem occurs, in part, because CLEENEX is unable to automatically optimize a data cleaning operation, i.e., choose the best algorithm to execute a data operation.

1.3 Objectives

The main goal of this thesis is to design and implement an optimizer to be integrated in the CLEENEX [15] data cleaning research prototype. This optimizer is able to find the graph of transformations that ensures the best trade-off between performance and quality of the results. For example, for the matching operator, the quality of the results are measured in terms of duplicates detected.

The optimizer needs to be able to optimize the graph as a whole, that is, it cannot act at each data transformation and optimize it individually and regardless of the remaining data transformations defined in the graph of transformations. To be able to find the set of transformations that together guarantee the least expensive graph of transformations, the optimizer includes a cost model. This cost model takes into account both the performance of each algorithm and its output quality.

1.4 Main Contributions

This thesis main goal is the implementation of an optimizer. To achieve that goal the following contributions were made:

- A cost model able to measure the cost of an algorithm. This cost takes into account the trade-off between the performance and output quality of each algorithm;
- An infrastructure that facilitates the addition of new algorithms. This also prepares CLEENEX to more easily implement distribution in its algorithms;
- Replace the generation of code in runtime for the matching operator to compile time;
- Experiments with Java native parallelism mechanisms for the matching operator;
- Experimental evaluation for the presented cost model and optimizations.

1.5 Document Outline

This document is organized as follows. Chapter 2 provides background about relational optimization and data cleaning concepts. In Chapter 3, we detail techniques to scale-up the deduplication task, one of the most expensive tasks of data cleaning, we describe research prototypes that distribute the matching execution, and finally we explain some data cleaning research prototypes concerned with the performance of a data cleaning process. The proposed solution to the problem addressed is detailed in Chapter 4. In Chapter 5, we evaluate the proposed solution. Finally, Chapter 6 presents the conclusions and future work.

Chapter 2

Background

Query optimization is a problem common to all Relational Database Management Systems (RDBMS) in the sense that they all aim at efficiently executing a user's query.

As mentioned in Section 1.2, the problem pursued in this work consists on optimizing the data cleaning process and its tasks over relational data. The optimizer will be integrated in the CLEENEX prototype. As said in Section 1.1, in CLEENEX, there is a separation between the logical and physical layers. RDBMS follow a similar approach, therefore, optimizations proposed for RDBMS can also be applied in the optimizer that we will create.

In Section 2.1, we make a review of relational query optimization, going through the steps of query processing, and several optimization techniques and algorithms. Section 2.2 introduces the main data cleaning concepts, focusing on a critical data cleaning task, the approximate duplicate detection, in Section 2.2.1.

2.1 Relational Query Optimization

When a user requests a query to be performed by a RDBMS, that query goes through several steps before returning the desired output, as depicted in Figure 2.1. This process is known as *query processing*. First, the user issues a SQL query that is parsed and translated into a relational algebra expression, and then further mapped into a tree-based structure, by the parser and translator module.

Every plan must identify the algorithm and indices that each tree node (a.k.a., operation) must use. The process of identifying the algorithm and indices to use in each node is known as annotation. An annotated node is called an *evaluation primitive*. An annotated tree (i.e., whose nodes are all annotated), also known as *query-evaluation plan* or *query-execution plan*, is what the execution engine accepts as input to compute the results of the submitted query.

The optimizer receives as input a relational algebra expression further mapped to a tree-based structure and creates several equivalent execution plans based on it. Among those plans, one of them is the most efficient in terms of resource consumption (e.g., CPU, memory, I/O) and this is the one delivered to the execution engine.

In this thesis, we shall focus on a relational optimizer, whose goal is to find a good strategy to run a query, i.e., as efficient as possible. Considering the relational schema shown in Example 1, Figure 2.2 illustrates the translation of an SQL query over the Employee and Department relations (Figure 2.2a) to a relational algebra expression represented in a tree-based structure (Figure 2.2b), that is an example of a valid input to the optimizer.

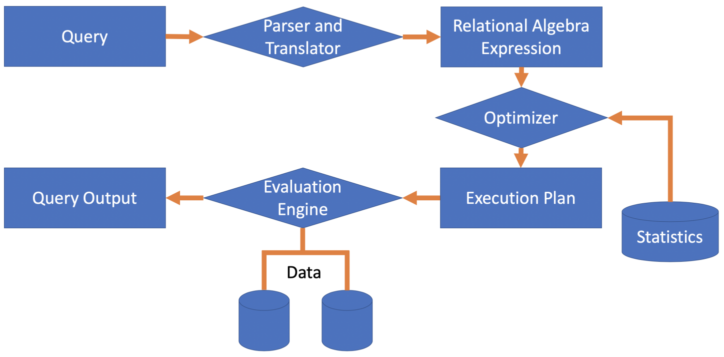


Figure 2.1: Relational database query processing steps (extracted from [28])

Example 1. Consider the following relational database scheme:

Employee(<u>emp_id</u>, name, address, salary, job, dept_id)

Department(<u>dept_id</u>, dept_name, num_employees)

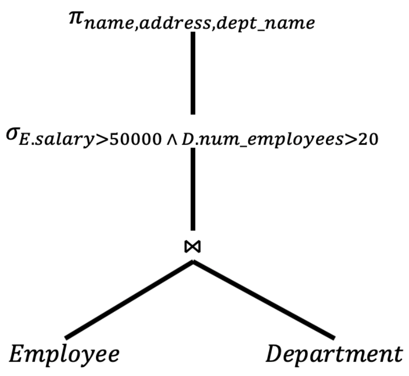
Project(<u>proj_id</u>, dept_id, admin_id, budget)

Where, *dept_id* and *admin_id* in Project are foreign keys to Department and Employee, respectively, and *dept_id* in Employee is a foreign key to Department.

Relational optimizers are typically classified as either rule-based, cost-based, or a mix of these two. *Rule-based* optimizers use a set of rules to transform a given execution plan into another possibly more efficient execution plan. *Cost-based* optimizers annotate the trees with different algorithms and indices, and manipulate the join order, to obtain a different query-evaluation plan. The most efficient plan is then selected.

```
SELECT name, address, dept_name
FROM employee E, department D
WHERE E.dept_id = D.dept_id
    AND E.salary > 50000
    AND D.num_employees > 20
```

(a)


$\pi_{name,address,dept\_name}$
$\sigma_{E.salary>50000 \wedge D.num\_employees>20}$
$\bowtie$
Employee
Department

(b)

Figure 2.2: Translation from SQL to a relational algebra tree

The query-evaluation plan performance is measured by its cost[1]. The *cost of a plan* is given by the sum of the cost of each node operation. Using the example of Figure 2.2, we need to perform a join, a selection, and a projection operation. To perform these operations, the optimizer needs to choose which algorithms it will use. Each algorithm has different costs and prerequisites (e.g., need to access a relation through an existing index). Concerning the selection operation, the algorithm choice is mainly dependent on the existence of an index. If there is not an index, we need to perform a table scan (i.e., searches the whole table) to find every record that satisfies the selection predicate. If there is an index, the chosen algorithm depends on the index's type (clustered, non-clustered, covering, etc), of the predicate's type (equality or comparison) and whether the predicate uses a key attribute or not. Regarding the join operation, it can either be a merge join, an indexed nested loop, a block nested loop, or another join algorithm. The indexed nested loop join, for example, is useful when there is an index on the join attribute of the inner relation (in our example, the Department's *dept_id*), whereas the merge join is appropriate when the relations are sorted by the join attributes.

Independently of the optimizer's type (rule-based, cost-based, etc), they all aim at choosing the best strategy (i.e., query-execution plan) to run a query. These optimizers have in common the fact that they need an internal data structure known as system catalog (explained in Section 2.1.1) to store statistics about the database status. In Section 2.1.2, we detail rule-based optimization, in Section 2.1.3, cost-based optimization. We detail a dynamic programming algorithm, the Join Reordering Algorithm in Section 2.1.4, and discuss Heuristics in Section 2.1.5. Finally, in Section 2.1.6 we describe an optimizer that combines the rule- and cost-based optimizer.

2.1.1 System Catalog

Relational database optimization techniques assume the existence of statistics about each database relation, as for example, its number of tuples, number of distinct values, etc. All relational database systems contain a structure known as *system catalog* that is responsible to store all these statistics, and also to maintain descriptive data about every table, index, and view (e.g., their names, structure, attributes, etc).

Relevant statistics available in the system catalog, for a given relation *r*, are: the number of tuples, the number of blocks/pages containing tuples of *r*, the size, in bytes, of an *r* tuple, the number of distinct values for a given attribute (or set of attributes), etc [28]. Regarding the indices, in the case of a B+-Tree, the catalog stores its depth and number of leaf pages. Additionally, most databases also store the distribution of values for each attribute as an *histogram*, that can be used, for example, to estimate a query's selectivity, i.e., how many records are retrieved by the query.

The estimations may not be the most accurate possible since the catalog is updated only in periods of low load. Hence, there may be some inconsistencies between what is stored in the database and what the catalog reports.

[1]Only cost-based optimizers have the notion of a query cost.

2.1.2 Rule-Based Optimization

Rule-based optimizers use a set of rules, known as *heuristics*, to transform a query plan into another, usually less expensive, and equivalent plan. A heuristic, by definition, is ”an approximation to the problem’s solution”, hence, sometimes, the application of a heuristic will result in a plan less efficient than expected, however, in most cases, it holds as true. There are several types of rules that can be used in an optimizer: constructive rules (as employed by Starburst [19]), and transformation rules (as employed by Volcano [18], and its successor, Cascades [17] - the basis of the SQL Server optimizer).

Constructive rules make use of existing integrity constraints and rewrite a query into semantically equivalent ones, a technique known as *semantic query rewriting* [21], by applying the integrity constraints to the conditions of the query `WHERE` clause. For example, assume that there is an integrity constraint $job = ''Programmer'' \rightarrow salary > 25000$. A semantic rewrite would turn the query in Figure 2.3a into the query in Figure 2.3b. If we assume that there is an index created on the salary attribute, then the execution of the query in Figure 2.3b will be faster. However, if there is not an index, then it is just a waste of time, hence, the existence of the index must be checked.

```
SELECT name, dept_name
FROM Employee E, Department D
WHERE E.dept_id = D.dept_id
    AND job=”Programmer”
```

(a)

```
SELECT name, dept_name
FROM Employee E, Department D
WHERE E.dept_id = D.dept_id
    AND job=”Programmer”
    AND salary > 25000
```

(b)

Figure 2.3: Example of a semantic query rewrite

Transformation rules [28], also known as *equivalence rules*, enable to transform a relational-algebra expression into an equivalent expression. The idea behind these rules is that these transformations will lead to a different but equivalent plan, possibly with a smaller cost. A widely-known equivalence rule expresses that theta-join operations are commutative, i.e., the order by which they are executed may be switched without any implication on the results. This equivalence rule can be expressed as follows: $A \bowtie_\theta B = B \bowtie_\theta A$.

2.1.3 Cost-Based Optimization

Cost-based optimization uses the plan’s cost to decide which execution plan is the most efficient. To estimate the plan’s cost, the optimizer uses the system catalog, which stores the current database status, while also taking into account the characteristics (i.e., the cost) of the algorithms (e.g., merge join, indexed nested loop join, etc) and the operators that are included in the plan.

Cost-based optimizers manipulate the joins order [28] to improve the plan’s cost. Without any optimization, for a join with n relations, there are $(2(n-1))!/(n-1)!$ possible combinations. Testing all these combinations is unfeasible for large values of n (e.g., for $n = 10$ there are 17.6 billion combinations). For example, $(R_1 \bowtie R_2 \bowtie R_3) \bowtie R_4 \bowtie R_5$ has 1680 possible orders. To reduce the number of combinations to test, one possible technique is to perform $(R_1 \bowtie R_2 \bowtie R_3)$ first, which has 12 combinations, then, we

```
procedure findbestplan(S)
if (bestplan[S].cost is not infinite)
/* bestplan[S] already computed */
return bestplan[S]
/* else bestplan[S] has not been computed
earlier, so compute it now */
if (S contains only 1 relation)
set bestplan[S].plan and bestplan[S].cost based
on the best way of accessing S
else for each non-empty subset S1 of S such that S1 != S
P1 = findbestplan(S1)
P2 = findbestplan(S - S1)
A = X
/* X = best algorithm for joining results of P1 and P2 */
cost = P1.cost + P2.cost + cost of A
if cost < bestplan[S].cost
bestplan[S].cost = cost
bestplan[S].plan = "execute P1.plan; execute P2.plan;
join results of P1 and P2 using A"
return bestplan[S]
```

Listing 2.1: Join Reordering Pseudo-Algorithm

join that result (represented as $R_{1\bowtie 2\bowtie 3}$) with the other two relations $(R_{1\bowtie 2\bowtie 3} \bowtie R_4 \bowtie R_5)$, and again, we have 12 combinations, thus reducing the total possible orders to 12 + 12, thus 24.

However, a certain join order may be optimal for a given operation, but suboptimal for another. We say that a particular sort order of tuples is in an *interesting sort order* if that specific order may be useful for later operations (e.g., in $R_1 \bowtie R_2 \bowtie R_3$, if you compute first $R_1 \bowtie R_2$ and order R_2 by $R_2 \bowtie R_3$ join attribute(s), then that sort order may minimize the merge join cost). For a set of *n* relations, there are 2^n interesting sort orders that must be saved.

2.1.4 Join Reordering Algorithm

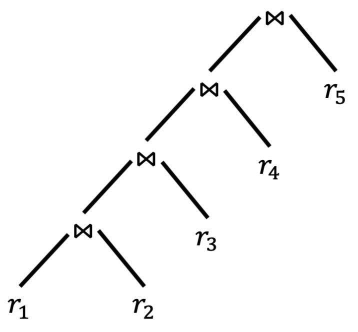
The *Join Reordering algorithm* is a dynamic programming algorithm. *Dynamic programming algorithms* are widely used by the RDBMSs to find the optimal query-execution plan by performing an exhaustive search. These algorithms construct all possible alternative query-execution plans. After creating a plan, its cost is evaluated. The plan with the least cost is saved to be used in the future if the same query is requested.

The join reordering algorithm may be performed iteratively, as the pseudo-algorithm in Listing 2.1 shows. At each iteration a new plan to perform the join is generated. If the new plan is cheaper than the currently saved plan, then it replaces it (i.e., it is saved), otherwise, it is discarded. If there is no join, i.e., the query only contains a relation, the best plan is defined as the best way of accessing that relation. After finding the cheapest plan, it is saved and returned every time the same query that triggered the algorithm is performed.

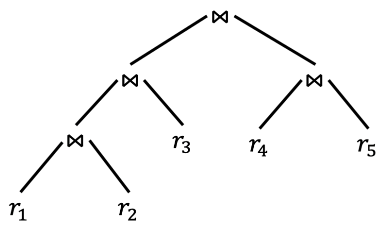
2.1.5 Heuristics

Applying the Join Reordering Algorithm (explained in Section 2.1.4) may not be enough to enhance performance. Dynamic programming algorithms perform an exhaustive search, generating many execution plans. To prune the number of plans that are generated, optimizers use heuristics. Applying certain heuristics can end up cutting out a good (i.e., efficient) execution plan. However, this is a risk that optimizers have to take.

A commonly used heuristic states that, in a tree, the join's right operand must be always a database relation, this way, all the execution plans that do not satisfy this condition will not be generated. Such query plans are called *left-deep join tree*, they are very convenient for pipelined evaluation (i.e., a node/-operation higher in the tree receives directly the results from its child instead of consuming the data from a temporary table that stores the results of its child). Figure 2.4 shows the difference between a left-deep join tree (Figure 2.4a) and a non-left-deep join tree (Figure 2.4b). This heuristic will sometimes find only a suboptimal plan since it does not test all possible join orders. However, it is more efficient than searching all possible join orders, especially if we use a dynamic programming algorithm, that allows us to store that plan for further use.



(a) Left-deep join tree



(b) Non-left-deep join tree

Figure 2.4: Examples of join trees

Other commonly applied heuristics are the predicate (selection) and projection pushdown. Here, the optimizer pushes the selections and projections as deep in the tree as possible, giving preference to perform first the selection, and just then the projection pushdown, since the first one has more probability to reduce the number of tuples to be used in the operations that follow in the tree. The selection pushdown is represented by the equivalence rule, $\sigma_\theta(R_1 \bowtie R_2) = R_1 \bowtie (\sigma_\theta(R_2))$, with θ being a subset of attributes of one of the relations (in this case, R_2). A similar rule exists for projection pushdown.

Many applications execute the same queries repeatedly, however, with different values for their constants. Yet another heuristic is to save the cheapest plan, found in the first time a query was run (with some constants values) for further uses, even though an optimal plan for certain constants may be suboptimal for others. This technique is called *plan caching*.

2.1.6 Cost-Based Optimization with Equivalence Rules

Reducing the number of tuples and columns that are involved in the intermediary operations (i.e., between the tree nodes) may help reducing the query's cost. That said, instead of working with the original query plan and just changing the algorithms and the joins order, it is useful to also change the query-evaluation plan to some other less expensive but equivalent plan. To accomplish that goal, we must use the techniques introduced in Section 2.1.2 along with the heuristics referred in Section 2.1.5. Just then, we can apply a cost-based optimization in order to find the best join algorithms and indices to use.

Searching equivalent query-evaluation plans is performed extensively, i.e., by applying equivalence rules while it is possible to generate new expressions. Less relevant plans are pruned, thus reducing the search space. Plans that are being evaluated, and whose cost is higher than the cheapest plan previously found, are considered irrelevant and are pruned.

2.2 Data Cleaning

Maintaining a database clean over the years is a difficult task. In fact, several people may insert data in a different fashion, leading to inconsistencies, and possibly, duplicates. For example, a user may enter misspellings, have different assumptions while inserting data (e.g., inserting ”J. Peralta” instead of ”Jake Peralta”), or she may ignore some business rules (e.g., sell a ticket to an underage). These data quality problems occur with a greater probability when there is no underlying schema (e.g., some schemes have an attribute for the first and last name, others only for the whole name, etc). Data quality problems (in particular, data redundancy) are even more noticeable when we intend at integrating heterogeneous data sources. Each data source may have a representation for the same real entity, data representations may contradict each other (e.g., in a data source the same person is 50 years old whereas in another 20), or have identical representations that refer to different real entities.

Figure 2.5 shows two data sources referring to TV Series characters (Figure 2.5a and Figure 2.5b). There are two entries that refer to the same ”real” entity (*star_id* 5 and *sid* 563) but: *(i)* they have different primary keys, and *(ii)* source A uses the character name whereas source B its nickname (”Thomas” and ”Tommy”, respectively). An identical case occurs with the first entry of both data sources (*star_id* 1 and *sid* 7). Also, whereas source A contains the character last name as ”Peralta”, source B wrongly contains the same last name as ”Peratla”. Using these two data sources, we can already conclude that: *(i)* we need to compare each pair of entries with a dictionary of synonyms, since ”Thomas Shelby” and ”Tommy Shelby” refer to the same real entity, and *(ii)*, another dictionary will be necessary to choose which representation has the correct character name (for the second case). String comparison techniques may fail at identifying that ”Thomas” and ”Tommy” are, in fact, the same entity, since they are written very differently. However, they should be able to identify the data quality problem in the second case, using the help of the dictionary to decide which one is the right name. The result of integrating these two data sources is exemplified in Figure 2.5c. There, the primary keys of each table were maintained to enable tracing to the original tables, a common practice when integrating data sources with different

star_id	first_name	last_name
1	Jake	Peralta
3	Amy	Santiago
5	Thomas	Shelby

(a) Favourite TV Stars

sid	name
7	J. Peratla
563	Tommy Shelby

(b) TV Stars

id	first_name	last_name	sid	star_id
1	Jake	Peralta	5	1
2	Tommy	Shelby	563	7
3	Amy	Santiago		3

(c) TV Stars (integrated and cleaned source)

Figure 2.5: Example of a process of integration of two data sources

schemes.

Data quality problems are usually solved, or at least minimized, by a data cleaning tool (examples of commercial data cleaning tools are Trifacta[2], Informatica[3], etc). Data cleaning tools can be divided in two categories: *(i)* transformation-based, or *(ii)* rule-based. In *(i)*, we have a graph of data transformations that are performed over dirty data. These data transformations are implemented by data cleaning operators, whose execution cleans data. Examples of transformation-based data cleaning tools are: the research prototype CleanM [16], , and the commercials Trifacta and Informatica. In *(ii)*, we define a set of rules (aka, *quality rules*) that the data must satisfy to be considered of good quality. If those rules are not satisfied, there is a violation. Violations must be repaired by a suitable data repair, usually, chosen between a set of heuristics. BigDansing [22] is an example of a rule-based data cleaning tool.

There are several expensive tasks in data cleaning, such as splitting, normalization, and approximate duplicate detection. However, the latter, also known as *record matching* (for a single pair of records) or *record-set matching* (for a set of pairs of records), is one of the most cumbersome tasks. Several techniques have been proposed to increase its performance and efficiency. Note that trivial record-set matching techniques perform the Cartesian product to obtain their results. Cartesian product is not feasible when dealing with large amounts of data, because the amount of resources that would be needed to perform that operation would be massive. In Section 2.2.1 we explain what is approximate duplicate detection, and common approaches to perform it. Later on, in Chapter 3, we discuss some techniques to *(i)* improve the approximate duplicate detection task performance in a single machine, and *(ii)* to distribute that task, by presenting some distributed join techniques based on a Map-Reduce approach.

2.2.1 Approximate Duplicate Detection

Approximate Duplicate detection is the problem of detecting that two tuples represent the same real entity, being one of the most expensive data cleaning tasks, since it demands every tuple to be compared with all existing tuples in a table (aka, Cartesian product), thus having a quadratic complexity. When we

[2]https://www.trifacta.com
[3]https://www.informatica.com

```
 if   similarity [name](A,B) < 0.8 return false
else  if   similarity [address](A,B) < 0.9 return  false
else  return  true
```

Figure 2.6: Pseudo-code of a matching rule

have several millions of tuples, performing a quadratic algorithm is undesirable, and in certain cases, unfeasible. A naïve approach to record matching uses string matching algorithms. It works as follows: *(i)* for each record, create a string that is the result of concatenating all fields of that record, *(ii)* to test the similarity between two records, take the string each one generated and compute their similarity using a string matching algorithm (e.g.: Levenshetein Distance, Jaro-Winkler distance, Soundex, etc). However, this naïve approach does not achieve good accuracy. Thus, other approaches were proposed to perform record matching. Most common approaches to record matching are: *(i)* Rule-based matching, *(ii)* Probabilistic matching, *(iii)* Learning-based matching, and *iv* Matching by Clustering.

Rule-based matching is the act of declaring that any pair of tuples *(x,y)* is either a match or not by applying a set of matching rules. These matching rules are similarity tests applied to one or more attributes, and can be seen as a set of ”if” and ”else” statements. Figure 2.6 illustrates the pseudo-code of a matching rule. There, we consider a pair of tuples *(x,y)* of the Employee relation a match, if the similarity score of their names is greater than 0.8 and the similarity of their addresses is greater than 0.9. The quality of matching rules is determined by their accuracy and coverage. Ideally, they should have high accuracy, i.e., classify correctly most of the pairs, and high coverage, i.e., cover a high number of pairs, however, usually, the greater the accuracy, the lower the coverage (i.e., the more specific the rule is).

To decide which rules will be applied in the decision-making process, one could follow two possible approaches: *(i)* manually generate matching rules, or *(ii)* generate rules through training data. The first approach demands that the user has previous knowledge of the data. The creation process is very cumbersome, demanding a very careful analysis of the data and also several attempts to obtain good quality matching rules. Therefore, it is an iterative, boring process for the user. The second approach uses machine learning to learn from data. We feed an algorithm with examples of true and false matches and it automatically creates several rules from those examples. Each rule that is generated has its coverage and accuracy measured, and only the one that has the best trade-off between both measures is selected. Then, that selected rule is expanded by the algorithm with new conditions (i.e., new rules).

In *Probabilistic matching*, the decisions are based on a set of variables over a probability distribution. A variable can be, for example, whether two employee names, of two different records, match. Or even if two records are a match or not. Although probabilistic techniques are known for their easy adaptability to the domain (e.g., matching employees), their are also known for their inefficiency.

Learning-based Matching and Matching by Clustering use machine learning to classify pairs of records as matches or non-matches. The *Learning-based Matching* approach uses supervised learning, i.e., there is the need to feed the machine learning algorithm with examples of true matches so that he can learn what are true matches in a give dataset. The *Matching by Clustering* approach enables a more independent solution since it is non-supervised, i.e., it can start classifying records right away.

Chapter 3

Related Work

In this chapter, we describe the most relevant works in what concerns the improvement of the performance of a data cleaning process, namely: *(i)* algorithms to scale up the approximate duplicate detection task, *(ii)* tools and algorithms to parallelize and distribute the approximate duplicate detection task, and *(iii)* data cleaning research prototypes.

In Section 3.1, we describe algorithms to scale up approximate duplicate detection. In Section 3.2, we detail an approach that addresses the parallelization and distribution of the deduplication task as a whole, and detail one algorithm that approaches the parallelization and distribution of the join (used to perform the Cartesian product). Finally, in Section 3.3, we detail the four most relevant data cleaning research prototypes that address the efficiency of a data cleaning process.

3.1 Scaling Up Approximate Duplicate Detection

Some techniques to improve approximate duplicate detection performance when faced with large amounts of data were proposed. Most of these techniques aim at creating clusters/blocks of records, limiting the pair comparisons only to those records inside the same block. These techniques are known as *indexing techniques* or *blocking techniques* [10]. Each record is associated to a *blocking key*. Records with the same (or, for some algorithms, similar) key value go to the same block and are compared. In this case, the key generation is a crucial step. It is the user's responsibility to define how the key is generated.

This section describes techniques to improve the efficiency of the approximate duplicate detection task. Section 3.1.1 describes a naïve optimization approach called *Traditional Blocking*, that creates clusters of similar records and performs only intra-cluster comparisons. The *Sorted Neighborhood Join* algorithm and some variations are described in Section 3.1.2. Section 3.1.3 explains the *Q-gram Based Indexing*, which assigns records with common *q*-grams to the same cluster, enabling a record to belong to several clusters. Section 3.1.4 describes the *Suffix Array Based Indexing*, similar to the *Q-gram Based Indexing* approach, but uses suffixes instead of *q*-grams. The *Canopy Clustering* technique is explained in Section 3.1.5. In Section 3.1.6, we summarize all algorithms detailed in this section.

3.1.1 Traditional Blocking

Traditional blocking [12] uses a user-defined key to put records that have exactly the same key value in the same block. The key is based on one or more attributes (e.g., the concatenation of the first two characters of every field). Once the blocks are created, the algorithm proceeds to the comparison phase. A Cartesian product is performed to generate all possible pairs of records found in a block. Then, the algorithm performs record matching to verify if a pair is a true match or not. Example 2 demonstrates how *Traditional Blocking* technique can be applied.

Example 2. Figure 3.1a shows five records from the Employee relation, whose schema was defined in Example 1. For this example, consider the blocking key to be the name attribute.

Since the blocking key values ”Tiago” and ”Tiago” are identical, the corresponding records go to the same cluster (Figure 3.1b). Analogously, ”Ana” and ”Ana” form another cluster, as shown in Figure 3.1c. The key value ”Rodrigo” has a cluster of its own since it does not match with any other key value (Figure 3.1d). Only clusters with more than one record need to perform record matching.

emp_id	**name(key)**
120	Tiago
237	Rodrigo
543	Ana
765	Tiago
775	Ana

(a) Initial data source

emp_id	**name(key)**
120	Tiago
765	Tiago

(b) Cluster 1

emp_id	**name(key)**
543	Ana
775	Ana

(c) Cluster 2

emp_id	**name(key)**
237	Rodrigo

(d) Cluster 3

Figure 3.1: Traditional Blocking example

The major drawback of the Traditional Blocking algorithm comes from the fact that it is too much dependent on the way the key is generated. For example, if there is an error on the key's value, two records that may be approximate duplicates will never be compared.

3.1.2 Sorted Neighborhood Join

Sorted Neighborhood Join (SNJ) [20] uses a different approach from the technique presented in Section 3.1.1. SNJ does not create blocks of records, instead, all records are sorted by the blocking key values and maintained in their original table (i.e., a single block). To mimic the underlying idea behind the blocks (i.e., limit the comparisons to records inside the same block), SNJ iterates through the records within a sliding window. A *sliding window* is a window with a fixed size w defined by the user. In the algorithm's first iteration, the window starts at the beginning of the table, and covers w records. At each iteration, record matching among the w records inside the window is performed. To proceed to a new iteration, the sliding window goes down one record (i.e., at iteration one, starts at record one, at iteration two, starts at record two, and so on). The algorithm finishes when the sliding window reaches the end of the table, that is, when the w^{th} record of the window is the last record of the table. SNJ does not guarantee that all true matches are captured, mainly because of the limitation of a fixed size sliding window[1]. Moreover, if we pretend to perform the approximate duplicate task with two tables that have the same schema, the algorithm creates an union of both tables. The SNJ algorithm works in the following three steps:

1. **Create key**: create a key for each record based on one or more attributes. In Figure 3.2a, analogously to Example 2, we select the employee name as key;

2. **Sort data**: sort the data source records based on the key defined in the previous step. Figure 3.2b shows the result of applying this step;

3. **Merge**: move a window with a fixed sized *w* (greater than 1 and less than the number of records) through the sorted records and generate all possible pairs of records inside that window. In Figure 3.2c, we defined a window of size 2. Only the records inside it will be compared. The window keeps moving until it reaches the end of the table.

With a sliding window of size two and a total of five records, the algorithm makes four iterations in the Merge phase. In the first iteration, it generates a pair with *emp_id* values 543 and 175, in the second one, *emp_id* values 175 and 237, in the third phase, *emp_id* values 237 and 123, and in the final one, *emp_id* values 123 and 120. If, for example, the window size was 3, the first iteration (of two) would generate the pairs [(543,175); (543,237); (175,237)][2]. All pairs generated in the Merge phase are then compared using matching rules, which decide if a pair of records is a match or not.

emp_id	**name(key)**
120	Tiago
237	Rodrigo
543	Ana
123	Thiago
175	Anna

(a) Creation of keys

emp_id	**name(key)**
543	Ana
175	Anna
237	Rodrigo
123	Thiago
120	Tiago

(b) Sorting data

emp_id	**name(key)**
543	Ana
175	Anna
237	Rodrigo
123	Thiago
120	Tiago

(c) Merging (iteration 1) - the bold lines represent the limits of the sliding window

emp_id	**name(key)**
543	Ana
175	Anna
237	Rodrigo
123	Thiago
120	Tiago

(d) Merging (iteration 2) - the bold lines represent the limits of the sliding window

Figure 3.2: Sorted Neighborhood Join example

[1]If two true matches are separated by more records than the window size, then they will never be compared, thus they are never considered as matches.

[2]Independently of the window size, the comparison is always performed between a pair of records.

The key chosen must include relevant information from the data source fields based on previous knowledge of the data. For example, a key could be created by selecting the first character of each field. Considering the two records $\{$1, "J. Peralta", "Angels Paradise Street", 55000, "Programmer", 1$\}$, and $\{$540, "Jake Peralta", "Angels Paradise Street", 70000, "Senior Programmer", 1$\}$, the first record has the key value *1JA5P1*, and the second has the key value *5JA7S1*. Although they refer to the same real entity, the way the user defined the key makes it hard (but not impossible, depending on the chosen *w* and number of records) for the SNJ algorithm to couple (and therefore compare) these two entries. If we performed a second passage with another key, the probability of finding similar records could improve. Moreover, given that real-world data is dirty, using only one way of generating key may prove to be insufficient to detect all possible matches.

To improve SNJ's accuracy, a *Multi-pass Approach* [20] was proposed. In a multi-pass approach, the SNJ algorithm is executed several times, each one with a different key. Moreover, with a multi-pass approach it is also possible to use *transitive closure* to find approximate duplicates. For example, if in the first pass, a given record r_1 is considered a duplicate of r_2, and in the second pass r_2 is considered a duplicate of r_3 then, by transitive closure, r_1 is also considered a duplicate of r_3. The final result of a multi-pass approach is the union of all the pairs discovered throughout the whole process, including those by transitive closure.

Several alternatives to the SNJ were proposed to improve its effectiveness and efficiency. The *Clustering Method approach* [20], uses a clustering algorithm to partition the initial records into independent clusters instead of depending on a user to define a key (SNJ's step 1). Then, the second and third setps of SNJ are applied inside those clusters. The clustering method extracts a key from each record based on one or more attributes, and the partitioning criteria is based on that key.

There may be several records with the same key, which, ideally, should be compared, since they may be approximate duplicates. In order for those records to be compared, they need to be covered by the same window. However, in large databases, it may not be possible to guarantee that the window covers all records with the same key, since the chosen window size w may be too small. The *Inverted Index Based approach* [20] deals with this problem by generating an inverted index whose index key is the unique blocking key values. Then, the index key values are sorted and the sliding window moves through the index key values rather than the blocking key values. In the sorted index key values list, each key appears only once. Each index key points to the records that have the same blocking key value. At each iteration of the algorithm (i.e., at each position of the window), the records that have the same blocking key as the index key being covered by the window, are verified (i.e., all possible pairs of records are generated and compared). In summary, in this algorithm, each window covers w index key values at the same time. Each index key value represents several records, thus, the probability of similar records being compared increases.

The *Adaptive Sorted Neighborhood* [32] overcomes one of the major problems of the original SNJ: having a fixed window size. When we have more records with a similar key than our window size *w*, there will be duplicates that are not detected. Since those records are never within the same window, they are not compared. The adaptive sorted neighborhood solves this problem by dynamically changing

emp_id	**name (key)**	**Bigram Sublists**	**Index Key Values**
120	Thiago	[th,hi,ia,ag,go], [th,hi,ia,ag], [th,hi,ag,go], [th,hi,ia,go], [th,ia,ag,go], [hi,ia,ag,go], [th,hi,ia], [th,hi,ag], [th,hi,go], [th,ia,ag], [th,ia,go], [th,ag,go], [hi,ia,ag], [hi,ia,go], [hi,ag,go], [ia,ag,go]	thhiiaaggo, thhiiaag, thhiaggo, thhiiago, thiiaaggo, hiiaaggo, thhiia, thhiag, thhigo, thiaag, thiaago, thaggo, hiiaag, hiiago, hiaggo, *iaaggo*
765	Tiago	[ti,ia,ag,go], [ti,ia,ag], [ti,ia,go], [ti,ag,go], [ia,ag,go]	tiiaaggo, tiaiaag, tiiago, tiaggo, *iaaggo*

Figure 3.3: Q-gram Based Indexing example

the window size w depending on the key's values. We start by creating a window at the beginning of the sorted table, then, the size of the window keeps increasing as long as sequential keys (which represent their records) are similar, according to a string similarity function. A window covers all records whose keys have a similarity between each other greater than a predefined threshold. A new window starts when two adjacent records have keys whose similarity is below that threshold.

3.1.3 Q-gram based Indexing

Q-gram based Indexing [5] tries to overcome the Traditional Blocking (described in Section 3.1.1) limitations by enabling records with similar key values to be put in the same block (in traditional blocking only those that were strictly equal would be put in the same block). As in the Traditional Blocking technique, the Q-gram based Indexing detects all possible approximate duplicates. The key must be defined by the user based on one or more attributes. Then, it is converted into a list of *q*-grams, that is, a list of substrings of length *q*. A total of $k = c - q + 1$ *q*-grams are generated, with *c* being the key's length (number of characters) (e.g., "Tiago" generates the bigrams ["Ti","ia","ag","go"], where *c* is equal to 5 and *k* is equal to $k = 5 - 2 + 1 = 4$). Then, the algorithm generates all possible combinations with the previously created *q*-grams. Each combination must have a minimum length of *l*, with $l = max(1, \lfloor k \times t \rfloor)$ (i.e., must use l of the previously created *q*-grams to create a new list), and $t \in [0, 1]$ a user-defined minimum threshold. Finally, each sublist of *q*-grams is transformed into an index key value through the concatenation of the *q*-grams (e.g., the bigrams ["Ti","ia","ag","go"] generate the index key value tiiaaggo).

In Figure 3.3 we exemplify the application of a 2-gram based indexing (i.e., $q = 2$) using two records. As in Example 2, we use the employee's name as key, and use the *emp_id* value to identify the records. For the key value "Tiago", there are $k = 5 - 2 + 1 = 4$ bigrams. With those bigrams, and assuming $t = 0.75$ and $l = max(1, \lfloor 4 * 0.75 \rfloor) = 3$, we create all possible variations with maximum length 3. They are listed under the *Bigram Sublists* column in Figure 3.3. For the same t, for the key "Thiago", with $k = 5$ and $l = 3$, we create all variations with a minimum length of 3, by varying the original bigram list (["Th","hi","ia","ag","go"]). The column *Index Key Values* in Figure 3.3 lists the concatenations of the bigrams stored in the *Bigram Sublists* column.

Each index key value can be seen as the name of a cluster. If two records have an index key in common, then they are put in the cluster identified by that index key. Intra-cluster comparisons are

emp_id	name (key)	Suffixes List
120	Thiago	Thiago, hiago, iago, ago, go
237	Rodrigo	Rodrigo, odrigo, drigo, rigo, igo, go
765	Tiago	Tiago, iago, ago, go

(a) Generating the suffixes

Suffix	emp_id
Rodrigo	237
Thiago	120
Tiago	765
ago	120, 765
drigo	237
~~go~~	~~120, 237, 765~~
hiago	120
iago	120, 765
igo	237
odrigo	237
rigo	237

(b) Inverted Index Table

Figure 3.4: Suffix Array Based example

performed to detect approximate duplicates among records beloging to the same cluster. In Figure 3.3, it is possible to see that the index key value *iaaggo* is common to the key values ”Tiago” and ”Thiago”, therefore, they will be put in the same cluster and thus will be compared. The key value ”Thiago” generated 15 distinct index key values, meaning that it can belong to up to 15 clusters.

3.1.4 Suffix Array Based Indexing

Suffix Array Based Indexing [2] is very similar to the *q*-gram approach, but it uses the key's suffix instead of the full key value. A total of $k = c - l + 1$ suffixes are generated (aka index keys), with *c* being the key's length (number of characters) and *l* the suffix minimum length. The index key values are then placed in an inverted index table, that maps each suffix (i.e., index key value) to the records that contain it.

In Figure 3.4, we assume $l = 2$, that is, we keep generating suffixes until they have only two characters, thus $k = 5$ suffixes will be generated for key ”Thiago”, $k = 6$ for key ”Rodrigo”, and $k = 4$ for key ”Tiago”, as illustrated in Figure 3.4a. Figure 3.4b represents the inverted index table that results from the previously generated suffixes. The lower the value of *l*, the higher the probability of several records having the same set of suffixes, thus penalizing the ability to distinguish between them and penalizing the performance (since more comparisons between records are performed). To maintain the *Suffix Array Based Indexing* approach as accurate as possible and with a reasonable performance, we must define a threshold for the maximum number of records that can have a given suffix. When surpassed, the suffix is discarded. This threshold is known as the maximum block size, *b*. In Figure 3.4 we assume $b = 2$, being that the reason why the suffix ”go” was removed. Each inverted index table entry acts as a cluster. As in the *Q-gram Based Indexing*, there are only intra-cluster comparisons.

3.1.5 Canopy Clustering

The *Canopy Clustering* technique [7] [26] groups the records into overlapping clusters, also known as *canopies*. As in the previous techniques, the comparisons are exclusively intra-canopy. The canopy clustering method uses two similarity measures, one to map records into the canopies, and another to

emp_id	name (key)	Token List
120	Thiago	[(Th,1), (hi,1), (ia,1), (ag,1), (go,1)]
765	Tiago	[(Ti,1), (ia,1), (ag,1), (go,1)]

(a) Generating the tokens

Token	TF	(emp_id, DF)
Th	1	(120,1)
hi	1	(120,1)
ia	2	(120,1), (765,1)
ag	2	(120,1), (765,1)
go	2	(120,1), (765,1)
Ti	1	(765,1)

(b) Inverted Index Table

Figure 3.5: Example of Canopy Clustering's first phase

compare the records inside each canopy.

In a first phase, we create a list with one or more tokens for each key value. A token can be a word, a character, or a *q*-gram. Then, for each unique token, we create an entry in an inverted index table, indicating which records contain that token. For the sake of the example, we shall consider that the tokens are the key's bigrams. These bigrams are generated as in the *Q-gram Based Indexing* approach, described in Section 3.1.3. Figure 3.5 exemplifies the application of the *Canopy Clustering* technique, using bigrams as tokens. The tokens in the token list shown in Figure 3.5a have the format *(token, document frequency)*. The document frequency is the number of times that a given token occurs in the key value under evaluation. In Figure 3.5a, we do not repeat any token (just as an example, for the word ”Paralelepípedo” the token ”le” would be repeated twice, thus its token list would have an entry ”(le,2)”). Figure 3.5b shows the inverted index table that results from the two lists of tokens. There, for each token, we have its Term frequency (TF), i.e., the number of records where that token appeared, and the list of records that have that token along with their document frequency.

In the second phase, using the inverted index table, we create the canopies and associate their corresponding records. There are two approaches to perform this second phase: *(i)* the *Threshold Based Approach*, and *(ii)* the *Nearest Neighborhood Based Approach*.

Threshold Based Approach

In the *Threshold Based Approach*, we need to define two thresholds: the loose threshold t_l, and the tight threshold t_t. This approach starts by putting into a set S all record identifiers (*emp_id)*. Then, it extracts from S a random record S_x. For each record S_i in S, that has, at least, one token in common with S_x (using the inverted index table to discover them), we apply a similarity function to compare it against S_x. If the similarity score between S_x and S_i is greater than t_l, then S_i is inserted into the S_x canopy. Furthermore, if the similarity score between S_x and S_i is greater than t_t, then S_i is removed from S, preventing S_i to be associated to other canopies[3]. After comparing all eligible records with S_x, we remove S_x from S, and select another random record. This process finishes when S becomes empty.

[3]One record can belong to several canopies. However, once a comparison results in a score higher than t_t, that record can no longer belong to new canopies.

Algorithm		User Input	Uses Key	Groups Records
Traditional Blocking		Key generator	Group records	Yes
Sorted Neighborhood Join	Base	Key generator	Sort records	No
		Window size		
	Multi-pass	Key generator	Sort records	No
		Window size		
	Clustering Method	Window size	Sort records	Yes
	Inverted Index Based	Key generator	Form indexes	Yes
		Window size	Sort indexes	
	Adaptive	Key generator	Sort records	No
		Threshold		
Q-gram Based Indexing		Key generator	Form indexes	Yes
		Q-gram length		
		Minimum index length		
		Threshold		
Suffix Array Based Indexing		Key generator	Form indexes	Yes
		Minimum suffix length		
		Maximum records with same suffix		
Canopy Clustering		Key generator	Form tokens	Yes
		Token generator		
		Thresholds or neighbor parameters		

Table 3.1: Summary of the algorithms explained throughout Section 3.1

Nearest Neighborhood Based Approach

In the *Nearest Neighborhood Based Approach*, we need to define two parameters, the number of records r that are removed from S at each iteration, and the maximum capacity of a canopy m, with $r < m$. As in the *Threshold Based Approach*, we start by selecting a random record S_x from S, being that record the centroid of the newly created cluster. For each remaining records S_i in S that have, at least, one common token with S_x (using the inverted index table to discover them), we apply a similarity function to compare them against S_x. The m nearest records are inserted in the S_x's cluster, and the r nearest records are removed from S along with S_x. The algorithm continues until S is empty. When compared to the *Threshold based Approach*, this approach allows us to know how many records a canopy has. However, because there is the need to define a maximum capacity for each canopy (m), some true matches may not fall under the same canopy when it is full.

3.1.6 Discussion

In Section 3.1, we have detailed several algorithms that enable to improve the approximate duplicate detection task performance. Each one of the five algorithms presented uses different approaches to achieve the same goal: more efficiency.

In Table 3.1 we have the algorithms described in Section 3.1. In column *User Input* we list the parameters that the user must provide to the algorithm. Column *Uses Key* tells us how the key is used in the algorithm. Finally, column *Groups Records* confirms if the algorithm groups the records in any way to compare them among each other.

As Table 3.1 shows, all algorithms need some input. With the exception being the Clustering Method since it automatically generates the key, they all need the user to specify the key generator, i.e., how the key will be extracted from a record. Mostly, this how this key generator is defined is what will affect the effectiveness of these algorithms.

The performance of the algorithms presented in Section 3.1 has been assessed by surveys [6, 33]. Although these surveys used different datasets, and different test environments (i.e., the number of records used, the hardware in which the algorithms performed, etc) they claim similar results. Firstly, it is clear that for large amounts of data the Q-gram Based Indexing is too slow, thus it is not suitable for this thesis. The better performers are the Sorted Neighborhood Join (all presented variants) and the Traditional Blocking algorithms. Regarding the quality of the results produced, the Inverted Index Based Sorted Neighborhood is claimed as the best. The Traditional Blocking achieves good quality results, being on-pair with the Threshold-Based Canopy Clustering and the Adaptive Sorted Neighborhood. The algorithms that achieve the best trade-off between performance and quality of the results are: *(i)* the Adaptive Sorted Neighborhood, *(ii)* the Traditional Blocking, and *(iii)* the Inverted Index Based Sorted Neighborhood.

3.2 Parallel and Distributed Data Matching

The Achilles tendon of the approximate duplicate detection task is the fact that it demands a Cartesian product when using a naïve approach. Even with the improvements in efficiency that blocking techniques achieve (discussed in Section 3.1), there is the need to perform millions of comparisons when there is large amounts of data. Therefore, a solution to parallelize and distribute the approximate duplicate detection task, or at least, its most resource demanding sub-task, the Cartesian product, is needed.

Figure 3.6 shows a typical workflow of an approximate duplicate detection program. In the matching phase we perform the Cartesian product, in the similarity computation we compute the similarity value for each pair of records, and in the match classification, we decide if we consider a pair a match or not.

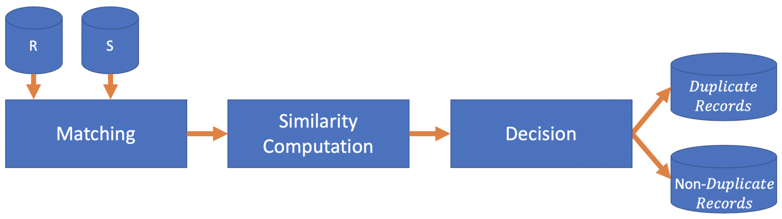


Figure 3.6: Approximate duplicate detection task workflow

In Section 3.2.1, we present Dedoop, a platform that focus on distributing and parallelizing the deduplication task, i.e., the whole workflow of Figure 3.6. Then, in Section 3.2.2, we present an approach which aims at optimizing the most expensive task of the approximate duplicate detection task, the Cartesian product, i.e., the Matching module in Figure 3.6. Finally, we conclude this section by summarizing both approaches in Section 3.2.3.

3.2.1 Dedoop

Dedoop [24] is a tool to perform efficient deduplication with Hadoop. Dedoop extends the blocking techniques detailed in Section 3.1, in particular the *Traditional Blocking* (Section 3.1.1) and the *Sorted*

Neighborhood Join (Section 3.1.2). The underlying idea is to increase performance, since Dedoop implements those techniques in a distributed setting, using the a framework based on the Map-Reduce (MR) paradigm [9], *Hadoop*. Hadoop uses a cluster of nodes to perform its map and reduce tasks.

Although blocking techniques provide an increase in performance when compared against the Cartesian product, they still demand too many pair-wise comparisons for large datasets. Most of these comparisons do not have dependencies among each other, i.e., to compare record r_1 with r_2, we do not need to know the result of any other comparison. Therefore, they can be performed in parallel. Performing record matching in parallel using the Map-Reduce (MR) paradigm has several advantages [10]: *(i)* we can quickly generate the results, thus enabling to evaluate the effectiveness of our algorithm and tune it, and *(ii)* if we take less time to obtain the results, we have a performance gain (i.e., the execution time of a data cleaning program is smaller).

A Dedoop workflow is composed by the following three jobs: *(i)* the Classifier Training job, *(ii)* the Data Analysis job, and *(iii)* the Blocking-based Matching job. The output of the first job is the input of the second one and analogously for the second and third jobs. Among these three jobs, only the latter is mandatory.

The *Classifier Training job* supports a machine learning-based match classification, i.e., instead of relying on users to define the matching rules, it uses a machine-learning algorithm to create the rules for a given dataset. Dedoop schedules a MR job to train a classifier based on previously labeled examples. The label in these examples is the similarity score between each training pair. The resulting classifiers are then saved in every node using the Hadoop's distributed cache mechanism.

The *Data Analysis job* exists to support load balancing strategies that Dedoop provides. The load balancing strategies are very important since data skew may exist, i.e., some blocking keys repeat much more than others, causing some blocks to have many more records than others. Due to this problem, sometimes the execution time is dominated by a single or few reduce tasks.

The *Blocking-based Matching job* is divided into three main steps: *(i)* blocking, using the Traditional Blocking and Sorted Neighborhood Join techniques, *(ii)* similarity computation, using string matching algorithms such as Levenshtein Distance and TF/IDF, and *(iii)* matching, where the matching rules are applied to make a decision. Figure 3.7 illustrates the Blocking-based Matching job workflow, with two input data sources, R and S. The *blocking phase* is a *map* task that uses a blocking technique (e.g., Traditional Blocking) to partition the dataset, delivering the generated blocks to the *similarity computation phase*, a *reduce* task. Each reduce task receives all records of a block (i.e., that have the same blocking key), and computes the similarity between each pair of records. Finally, after the similarity computation, each pair is classified as a match or non-match in the *match classification phase*, which occurs in the same reducer as the similarity computation phase. The resulting dataset M stores the matches between the input data sources R and S.

Dedoop is concerned about the efficient use of a cluster of MR nodes. There are two main efficiency problems when using blocking-based techniques: *(i)* data skew, which leads to load imbalances, and *(ii)* comparisons of the same pair of records in different nodes (i.e., repeated comparisons). When data skew exists, some of the available nodes may be blocked with huge partitions, delaying the job

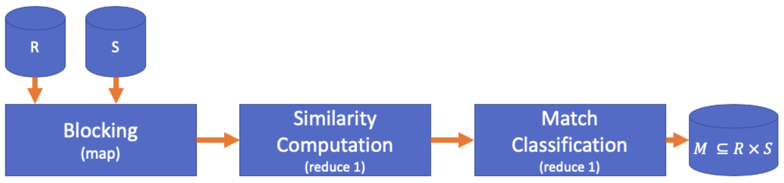


Figure 3.7: Blocking-based Matching job workflow

emp_id	**name(key)**	**last_name**
120	Tiago	Luiz
237	Rodrigo	Leite
543	Ana	Rodrigues
765	Tiago	Estradas
775	Ana	Félix
123	Tiago	Moniz
876	Ana	Galvão
645	Ana	Rodrigues
345	Tiago	Luis
132	Tiago	Luiz

(a) Initial dataset

Block	**Map1**	**Map2**	**Size**	**Pairs**
Ana	2	2	4	6
Tiago	2	3	5	10
Rodrigo	1	0	1	0
Total				16

(b) Block Distribution Matrix

Figure 3.8: Example of a Block Distribution Matrix

conclusion. The second problem arises when we use a blocking technique with a multi-pass approach. In the remaining of this section, we detail the technique used by Dedoop to perform load balancing, in order to deal with the data skew problem, and a technique to perform redundant-free comparisons to deal with the problem of repeated comparisons.

Load Balancing

When using the MR paradigm the load balancing problem appears when we forward work to reducer tasks. MR is able to perform load balancing by itself among the *map* tasks, however it is the programmer or platform responsibility to perform load balancing among the *reducer* tasks. To enable load balancing, Dedoop uses the Data Analysis job to create a data structure known as *Block Distribution Matrix* (BDM) which will be delivered to every *map* task of the Blocking-based Matching job. Then, each *map* task uses that data structure to perform the load balancing among the reducers.

The BDM shows the distribution of records per each blocking key in each map task of the Blocking-based Matching job. Moreover, it has also information about the number of records for a given blocking key (in all map tasks) and the number of pairs it is possible to generate for a given blocking key (i.e., number of comparisons that are performed). For example, using the dataset of Figure 3.8a, with two *map* tasks in the Matching job, and using the name as blocking key, the resulting BDM would be that of Figure 3.8b.

To perform load balancing, the *map* tasks of the Blocking-based Matching job take into account the information that the BDM has, and also the blocking technique that is being performed. On Traditional Blocking, Dedoop's only concern is distributing the blocks evenly among all reduce tasks. For example, in the BDM shown in Figure 3.8b, it will be performed a total of 16 comparisons. If we have two reduce tasks, each one performs 8 comparisons. However, on Sorted Neighborhood Join, there is the notion of a sliding window. If we consider each window a block (which is assigned to a *reduce* task), then there are records that are in more than one block. Therefore, there is the need to replicate records among reducers. Table 3.2 shows how the dataset of Figure 3.8a would be distributed among two reducers ($r = 2$) using when we perform a Sorted Neighborhood Join in Dedoop with a window size of three ($w = 3$). The i^{th} entity goes to reducer $\lfloor i \times \frac{r}{n} \rfloor$, with n being the number of records. Moreover, the last $w - 1$ records of reducer i are replicated in reducer $i + 1$.

Number (*i*)	**emp_id**	**name(key)**	**reducer_id**
0	775	Ana	0
1	543	Ana	0
2	645	Ana	0
3	876	Ana	0/1
4	237	Rodrigo	0/1
5	123	Tiago	1
6	120	Tiago	1
7	132	Tiago	1
8	765	Tiago	1
9	345	Tiago	1

Table 3.2: Distribution of records per reducer in Sorted Neighborhood Join

Redundant-free Comparisons

As discussed in Section 3.1.2, multi-pass approaches are necessary to deal with the dirty nature of real-world data. However, these approaches commonly lead to overlapping blocks, i.e., pairs of records with more than one common blocking key (e.g., in a multi-pass approach, if record A has the key values x, y, z and B has the key values w, y, z, then they have two overlapping keys, y and z, thus being compared twice). Therefore, when the block distribution is performed, and a pair of records is repeated across multiple blocks that go to different reducers, none of the reducers knows that *(i)* another block has a similar pair, and *(ii)* a repeated pair has already been computed in another reducer. Dedoop is able to limit the comparison of each pair to only and exactly one comparison. To do so, it adds to the traditional map output (blocking_key, record) in the matching job, an annotation containing a set with all record's blocking keys that are smaller (lexicographically) than the blocking key value under evaluation (e.g.: if we have two passes and we are on the second one, and if the blocking key value for a record in pass 2 is $Rosalina$, then that is the key under evaluation). Moreover, the pass number is added as a prefix to every blocking key (e.g.: in pass 2, blocking key value $Amadeus$, is turned into $2_Amadeus$). A reducer only compares two records if and only if the two sets are disjoint. Table 3.3 shows the signatures (i.e., the set of blocking keys of all passes) that are generated, for a subset of the dataset in Figure 3.8a, when there are two passes: one whose blocking key is the name and another whose key is the last name. With Traditional Blocking, for records number 0 and 4 the *map* task outputs the pairs $(1_Tiago, [0,])$ and $(1_Tiago, [4,])$, respectively. However, since we have two passes, another set of pairs will also be generated. For those same records, the *map* task outputs $(2_Luiz, [0, 1_Tiago])$ and $(2_Luiz, [4, 1_Tiago])$, respectively. Since both records 0 and 4 have common signatures, only the *map* task that is responsible

Number	*emp_id*	name	last_name	Signatures
0	120	Tiago	Luiz	{1_Tiago, 2_Luiz}
1	237	Rodrigo	Leite	{1_Rodrigo, 2_Leite}
2	543	Ana	Rodrigues	{1_Ana, 2_Rodrigues}
3	775	Ana	Félix	{1_Ana, 2_ Félix}
4	132	Tiago	Luiz	{1_Tiago, 2_Luiz}

Table 3.3: Two-pass blocking with keys being the name (1st pass) and last_name (2nd pass)

to evaluate the smallest blocking key value will perform the comparison.

3.2.2 Parallel Set-Similarity Joins

Some tools focus on the parallelization and distribution of the whole approximate deduplication task (e.g., Dedoop, described in Section 3.2.1). However, we can also focus in distributing and parallelizing the most expensive part of the approximate deduplication task, the Cartesian product. A solution to efficiently perform parallel set-similarity[4] joins using Map-Reduce (MR) was proposed by R. Vernica et al. [29].

The main difficulty when performing set-similarity joins using MR is to decide how data should be partitioned and replicated across the MR nodes. The way partitioning and replication is performed impacts the memory consumption and performance of the MR tasks. By default, the MR framework hash-partitions the data across the nodes based on key values (i.e., records with the same key value go to the same node). Typically, the key value is the joining attribute (i.e., the whole string, independently of its length). The proposed solution does not use directly the joining attribute value. Instead, it creates one or more signatures generated from the joining attribute value. These signatures are known as *partitioning keys*. Similar join attribute values should have at least one signature in common. Therefore, records with signatures in common are compared. An example of a signature is composed of the tokens of a string (e.g., ”This is my thesis” has 4 word-based signatures, ”This”, ”is”, ”my”, ”thesis”). To decrease the number of signatures each joining attribute value has, instead of using all signatures generated, the algorithm sorts all signatures and uses only a limited number of them. The number of signatures used by each joining attribute value is defined as a parameter of the algorithm.

The authors propose two algorithms, one for the self-join, and another for R-S join (i.e., between two different tables). The *self-join* algorithm starts by using the Basic Token Ordering (BTO) [29] algorithm to extract the signatures from the records and compute their occurrence frequency, in relation to the whole dataset. The output of this stage is an ordered list of tokens, ordered from the least used to the most used (e.g., for a dataset with records *(1, ABC)* and *(2, BC)*, the output is [B,C,A]). This output, in conjunction with the original dataset is given as input to the second stage, known as *Indexed Kernel*. This stage uses an algorithm called PPJoin+ Kernel (PK) [29, 31] to perform the comparison between records. The comparisons are not performed among all records. Instead, each signature of each record is assigned to a group represented by a synthetic key (e.g.: for record *(2,BC)*, the algorithm may assign

[4]Set-similarity join refers to the task of finding all pairs of records from two relations whose pair similarity score is higher than a given threshold.

key value X to signature B and key value Y to signature C). Only records with the same synthetic key are compared. To avoid data skew, these synthetic keys are assigned in a round-robin fashion (i.e., there is a predefined set of synthetic keys that are assigned to signatures). In the third stage, the record join is performed using the Basic Record Join (BRJ) [29] algorithm. The output of this final stage is the concatenation of those records whose similarity score (computed in the second stage) is greater than a threshold.

To enable this algorithm to be used in a R-S join scenario, the authors changed the second and third stages, since now there are records from two datasets as input, i.e., two input streams. This is a challenge because the Map-Reduce framework was originally designed to accept a single input stream. To distinguish between two different input streams, the algorithm adds a tag with the identification of the corresponding input stream when processing the records (e.g., the first stage would receive a record *(R, 1, ABC)*, i.e., the record *(1, ABC)* belongs to dataset *R*).

More recently, this algorithm was included in a new platform developed by some of the authors of this algorithm, the AsterixDB [3, 23]. AsterixDB is a database management system for Big Data. In addition to allowing to do everything that a common database allows, such as storing data, queries, it also added the possibility of performing operations such as deduplication. For that it uses the algorithm described in this section.

3.2.3 Discussion

In this section we presented two data cleaning tools with different purposes. Dedoop, detailed in Section 3.2.1, optimizes and distributes the deduplication task as a whole. The solution proposed by Vernica et. al., described in Section 3.2.2, focus in the optimization and distribution of the set-similarity join, that is, the task of discovering all pairs of two or more datasets whose similarity is greater than a given threshold.

Dedoop takes advantage of the vastly studied blocking algorithms which is seen as a benefit of this solution since it applies techniques that are easily understood and known by the community. However, at this moment, it only enables two blocking algorithms, Traditional Blocking and Sorting Neighborhood Join. It provides for both algorithms a multi-pass approach which is beneficial to improve the quality of the results (as discussed in Section 3.1.2. We see the restricted number of blocking algorithms as a limitation of the solution, mainly because of the three blocking algorithms identified as the best performers in Section 3.3.5, it only uses the second one, the Traditional Blocking (Section 3.1.1). The best performer, the Adaptive Sorting Neighborhood Join (ASNJ), detailed in Section 3.1.2, is discarded as well as the third best performer, the Inverted Index Based Sorted Neighborhood (IBSN), also detailed in Section 3.1.2. Although implementing the ASNJ would greatly increase the communication cost, which may be a reason to not implement that algorithm in a distributed setting, we do not see a valid reason why Dedoop does not support IBSN. Supporting more blocking algorithms would increase Dedoop's usability, since other data cleaning platforms could use Dedoop just to distribute the blocking algorithms. However, most data cleaning platforms have a wider range of supported blocking techniques than Dedoop. Regarding the solution proposed by R. Vernica et. al., we consider it a good solution to overcome the cartesian product costs, however, it is limited to that task. It does not perform the deduplication task as Dedoop. Moreover, both solutions have a big memory footprint, since they need to maintain state between the map-reduce jobs. For example, Dedoop needs to maintain a copy of the Block Distribution Matrix (BDM), which enables load balancing, in each *map* task. Since the BDM has one entry for each blocking key, when we have a big dataset that may be a problem. The solution by Vernica also has to maintain a copy of the ordered list of tokens, computed in the first map-reduce job, in each *map* task of the second map-reduce job (where the similarity is computed).

3.3 Data Cleaning Research Prototypes

In this section, we present the data cleaning research prototypes that, as far as the author is aware of, address the efficiency of a data cleaning process. For each prototype, we detail its architecture, how it handles large amounts of data, the approaches taken to deal with expensive tasks, and the platform specificities for solving the efficiency problem of a data cleaning process.

In Section 3.3.1, we present the CLEENEX prototype, a transformation-based data cleaning tool. Then, in Section 3.3.2, we detail the CleanM prototype, a recently proposed transformation-based data cleaning tool. Section 3.3.3 details the rule-based data cleaning tool BigDansing. The last prototype, the cross-platform data processing tool RHEEM, is explained in Section 3.3.4. Finally, in Section 3.3.5, we provide an overview over the four prototypes explained in this section.

3.3.1 CLEENEX

CLEENEX [15] is an extension of the data cleaning tool AJAX [14]. CLEENEX incorporates user feedback into a data cleaning process, introducing the notion of Quality Constraints and Manual Data Repairs. It is a transformation-based data cleaning tool that provides a specification language, that is an extension of the SQL language, for describing data transformations. Similarly to AJAX, CLEENEX provides a separation between the logical level, where the developer defines through an SQL-like syntax, the data cleaning program, i.e., the sequence of data cleaning operators to be applied, and the physical level, that describes the implementation of a data cleaning program, i.e., which algorithms shall be used to execute those operators. This separation opens optimization opportunities that can be applied at the physical level.

CLEENEX introduces the notion of a Data Cleaning Graph (DCG) to represent a data cleaning program, i.e., the workflow of data transformations to be applied to a dataset. A DCG is a Directed Acyclic Graph (DAG) whose nodes represent the transformations that shall be performed, or the relations that serve as input for those transformations. The edges connect relations to data transformations. Each transformation can be specified through one of the five logical operators supported by CLEENEX: *(i) mapping* which takes a single relation as input and outputs one or more relations, *(ii) view*, an operator that represents a simple SQL query augmented with some integrity checking over the output relation, *(iii) matching*, which applies an approximate join to two input relations to detect approximate duplicate records, *(iv) clustering*, that takes a single input relation and groups its records according to a given clustering algorithm (e.g., transitive closure), and *(v) merging*, which groups the input records according to some grouping attributes and collapses each group into a single tuple using a user-defined aggregation function. To complement these operators, CLEENEX supports User Defined Functions (UDFs), implemented in Java, enabling them to be invoked within operators. These UDFs must be registered in the CLEENEX functions/algorithms library. All the relations involved in a DCG (input and intermediate relations generated by the graph) are stored in a RDBMS.

As mentioned earlier, CLEENEX introduces *Quality Constraints* (QCs) and *Manual Data Repairs* (MDRs). These QCs and MDRs are defined over the set of input and output relations that compose the DCG. Each relation can be associated to a set of QCs that its records must satisfy (e.g., the QC $qc_1 : salary > 600$ defines that the salary attribute value must be higher than 600€). A QC is a mechanism to call the user's attention for tuples that do not satisfy certain conditions. A *blamed tuple* is a record that does not satisfy a QC. Every QC has a table for its blamed tuples. A MDR may be defined over any relation of the DCG consisting in a updatable view and an action. The view defines the set of tuples that the user sees when an MDR is executed. An action can be an update, insertion or removal. When a QC is defined over a relation, an MDR can also be defined over the same relation to provide a way for incorporating a user action to manually correct the blamed tuples.

The CLEENEX architecture is illustrated in Figure 3.9. It is composed by the following nine components:

1. *Parser:* performs the syntactical analysis of the data cleaning program, generates the DCG, and generates the Java code needed to execute the data cleaning program (including the QCs and MDRs);

2. *Catalog Manager:* saves the Java representation of the DCG;

3. *Database Manager:* communicates with the Relational Database Management System (RDBMS). Upon *Catalog Manager* request, during compilation time, it creates the output tables for each data transformation. On execution time, it receives requests from the *Scheduler*, the *Debugger*, or the *Graphical User Interface* (GUI) to execute SQL queries;

4. *Quality Constraint (QC) Manager:* responsible to create the table that will store the blamed tuples for each QC;

5. *Manual Data Repair (MDR) Manager:* constructs and applies the MDRs to the relations according to the user feedback;

6. *Scheduler:* executes the compiled data transformations according to the order defined in the data cleaning program;

7. *GUI:* graphical representation of the data cleaning graph (DCG);

8. *Debugger:* enables the user to trace the execution flow of a DCG;

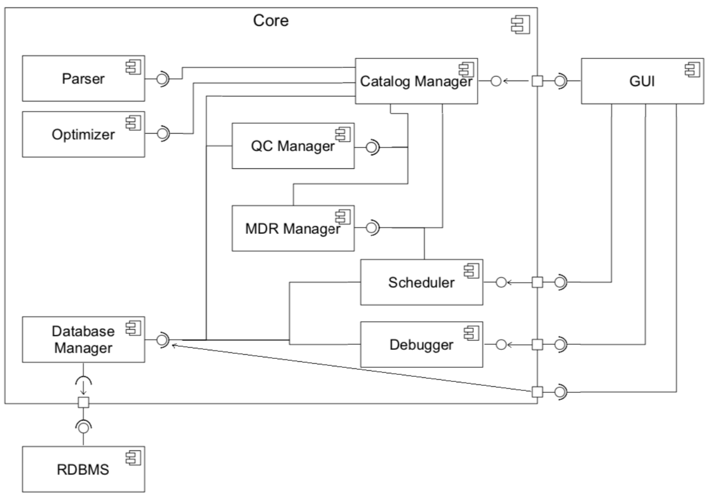


Figure 3.9: CLEENEX component architecture

9. *Optimizer:* responsible to choose the best physical execution algorithms for each logical data transformation in the DCG and to optimize the DCG as a whole.

In the current version of CLEENEX, there are two optimizations defined: *(i)* push down to the RDBMS all operations that are possible to perform there (e.g. the View operator), since an RDBMS already defines several optimizations, thus taking out the burden of the CLEENEX optimizer, and *(ii)* it allows the user to provide hints to the optimizer to use a given algorithm to perform the approximate duplicate detection task.

3.3.2 CleanM

CleanM [16] is a recently proposed data cleaning tool that aims at unifying the most popular data cleaning operations into a single tool. CleanM allows the users to express several data cleaning tasks such as denial constraints, deduplication, data transformations (e.g., merging columns of a dataset), and term validations. Furthermore, it supports an extension of the SQL language that enables to specify those cleaning tasks. Data cleaning operations are firstly optimized, and then deployed in a scale-out fashion, using frameworks such as Spark.

CleanM proposes a three-level optimization for a given data cleaning program as represented in Figure 3.10. First, the *Parser* transforms the data cleaning program into an *Abstract Syntax Tree* (AST), as in an RDBMS. The AST is further mapped by the *Monoid Rewriter* into an optimizable and inherently parallelizable calculus, the *monoid comprehension calculus* [11]. This calculus is able to represent complex operations between different data collection types (e.g., JSON, relational, etc) in a unified way, thus enabling to optimize the task as a whole. Moreover, some optimizations, such as filter pushdown (detailed in Section 2.1.2), are applied to the comprehensions. Then, at the second-level optimization, the comprehensions generated are translated into an intermediate algebra, the *nested relational algebra* [11], by the *Monoid Optimizer* module. This intermediate algebra has three major benefits: *(i)* it defines a set of rules, removing any query nestings, which in data cleaning programs, constitutes a major concern, *(ii)* independently of the data source, or the desired operation(s), all monoids are translated into the algebra, enabling the detection of intra- and inter-operator optimizations (e.g., work/data sharing between operators, i.e., if task A performed operation X, and task B also performs operation X but with different parameters, then the tasks are merged - this is known as coalescing operators), *(iii)* since the comprehensions are being translated into an algebraic form, optimization techniques vastly studied in the context of relational algebra can be used. Finally, the third-level of optimization is the mapping of the algebraic operators to a physical plan, which is able to deal with common problems such as data skew. This third-level of optimization is performed by the *Plan Rewriter* module. Independently of the complexity of a data cleaning task and the data sources, CleanM treats the whole task as a single query, optimizing it as a whole. Completing the optimization steps, the physical plan is translated into code by the *Code Generator* in order to execute the data cleaning program in a distributed execution engine (e.g.: Spark).

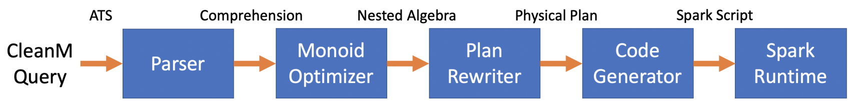


Figure 3.10: CleanM example workflow. Adapted from [21].

By using the *monoid comprehension calculus* and the *nested relational algebra*, CleanM is able to perform optimizations both at the logical and physical level. At the logical level [11], there are optimizations such as the aforementioned filter pushdown, performed by the *Monoid Optimizer* module. At the physical level, CleanM defines algorithms to deal with costly cases such as similarity joins and self-joins. Naïve implementations of such tasks use the Cartesian product, i.e., all-to-all comparisons. CleanM also uses blocking techniques (studied in Section 3.1). The length of the strings (i.e., the blocking key values) affects the number of blocks created, therefore, depending on that length, CleanM uses one of two techniques: *(i)* token filtering for smaller strings, using a similar approach to the already detailed Q-gram Based Indexing (Section 3.1.3), or *(ii)* clustering for bigger strings. More precisely, to perform clustering, CleanM uses a modified machine learning clustering algorithm, k-means. Note that k-means is an iterative algorithm, however, iterative algorithms and large amounts of data are usually not a good fit. Therefore, CleanM uses a single iteration k-means [27] to avoid performing multiple iterations, thus being able to achieve a better performance and execution time.

Unlike most data cleaning tools, CleanM treats all data cleaning operations as first-class citizens, instead of relying on black-box UDFs (i.e., UDF that are not defined by the language). For example, CLEENEX (detailed in Section 3.3.1) relies on black-box UDFs to perform and/or assist data cleaning transformations. By treating data cleaning operations as first-class citizens (i.e., UDFs whose optimizer knows how they behave), CleanM is able to perform a static analysis, i.e., to compute the cost of performing an operation at compile time, thus being able to perform a better and more realistic analysis of a data cleaning program cost (as defined in Section 2.1).

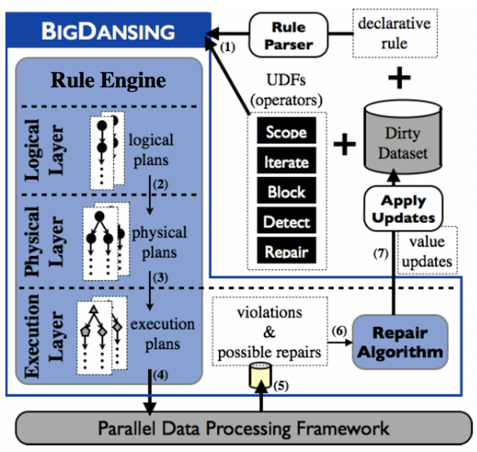


Figure 3.11: BigDansing architecture. Adapted from [22].

3.3.3 BigDansing

BigDansing [22] is a Big Data Cleansing tool that tackles the problem of efficiency and scalability. *BigDansing* differs from the previous approaches described so far in Section 3.3 because it is a rule-based data cleaning tool. A rule-based tool first detects which (pairs of) records violate a set of predefined rules. Then, either automatically or by asking for user's assistance, fixes the detected violations[5].

The *BigDansing* system architecture is illustrated in Figure 3.11. It is possible to distinguish two big modules (the ones in blue). The left-most is the *Rule Engine* module. It receives as input a dirty dataset and a *BigDansing job*, i.e., a rule expressed declaratively or procedurally (i.e., through the definition of UDF-based operators). Then, it outputs a set of violations and possible repairs. A BigDansing job defines which operations must be performed, and their order. The Rule Engine module is divided in three layers: *(i)* logical layer, *(ii)* physical layer, and *(iii)* execution layer. The logical layer is where we define a rule, independently if it is expressed declaratively or procedurally. At the physical layer, the logical plan with logical operators, built in the previous layer, is converted to physical operators and the whole plan is optimized. Finally, at the execution layer, the physical operators are mapped to the operators of the framework to be used (e.g., Spark, DBMS, Map-Reduce based, etc). The remaining module is the *Repair Algorithm*, which repairs the detected violations using the *Equivalence Class* algorithm [4, 8]. A BigDansing job can be defined using five logical operators: *(i) Scope*, that reduces the quantity of data to be treated, i.e., acts as a filter, *(ii) Block*, which groups records by a given key, *(iii) Iterate*, that enumerates all possible combinations of records in each block, i.e., performs the Cartesian product inside each block, *(iv) Detect*, which verifies if there is a violation for each pair, and *(v) Repair*, also known as *GenFix*, that retrieves possible solutions for the violation that was found. If it is a procedural rule, then it is up to the user to define the order by which operators are executed. If it is a declarative rule, then the *Rule Parser* module automatically translates it to a BigDansing job using the aforementioned logical operators.

The logical plan created in the logical layer, that represents the BigDansing job to be executed, is

[5]Sometimes a violation cannot be fixed, however we consider a dataset as cleaned if it does not have violations, or if it has only violations that cannot be repaired.

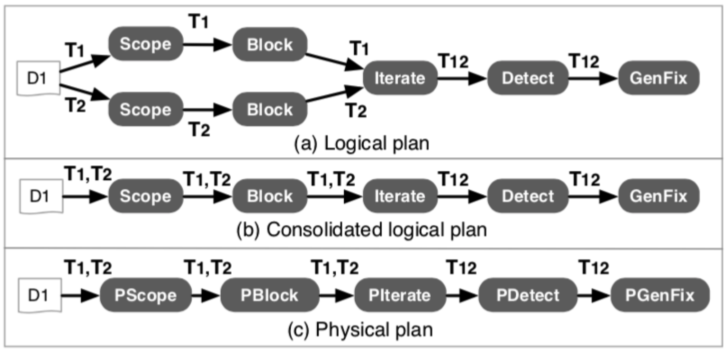


Figure 3.12: Plans for DC1. Extracted from [22].

optimized and translated into a physical plan composed of physical operators. There are two types of physical operators: a *wrapper*, which performs the operation demanded by a logical operator (*PScope*, *PDetect*, *PGenFix*, etc), adding physical details such as the input dataset, and an *enhancer*, that replaces a wrapper whenever there is an optimization opportunity. The enhancer is an optimized physical operator.

Possible enhancers are: *CoBlock*, *UCrossProduct*, and *OCJoin*. The *CoBlock* operator acts as a GROUP BY clause, that is, it groups all records with the same key (attribute) in the same group. The *UCrossProduct* performs an optimized self-join when the order of comparisons does not matter (i.e., when the join condition uses the $=$ or $\neq$ operators). If the order of comparisons matters (i.e., when we use the $<$, $>$, $\leq$ or $\geq$ operators in the join condition), then we use the *OCJoin* enhancer, which optimizes the naïve self-join.

Before a physical plan is generated, at the physical layer, BigDansing performs a static analysis, known as *plan consolidation*. Once the plan consolidation is finished, it outputs a consolidated logical plan. In the plan consolidation phase, the goal is to coalesce operators as much as possible. Coalescing can be performed when there are two identical logical operators that read the same dataset. Then, the consolidated logical plan is transformed into a physical plan. In this physical plan, wrappers will be replaced by enhancers whenever it is possible. Consider the denial constraint $DC1 : \forall T_1, T_2 \in D1, \neg(t1.name = t2.name \wedge T1.address = T2.address \wedge T1.job \neq T2.job)$, in which there is a violation if two employees have the same name and address and different jobs. Consider that the dataset *D1* refers to the Employee relation. For DC1, the logical plan in Figure 3.12a is generated. First, for T_1, we read the dataset, select the relevant attributes (emp_id, name, address and job) and perform the block operation. The same operations, with the same exact order are performed for T_2, thus instead of reading the dataset twice, we coalesce these operations into one, as shown in Figure 3.12b, generating the consolidated logical plan. Finally, we transform each logical operator into a physical operator (a wrapper or an enhancer), creating the physical plan illustrated in Figure 3.12c.

Both *wrappers* and *enhancers* have the same purpose: to implement a logical operator. When executed, they produce the list of violations to the rules that compose a data cleaning program. Once the records that do not satisfy the rules are found, they need to be repaired. BigDansing implements a widely used distributed repairing algorithm, the *Equivalence Class* algorithm [4, 8], unlike to most data cleaning rule-based tools, that use a centralized approach to avoid inconsistencies when applying repairs. The *Equivalence Class* algorithm first groups all records that should be equivalent together, and then decides how to assign values to each group. An equivalence class consists of pairs of the form (t, A), where t is a record, and A is one of t's attributes. In a dataset D, each record t and each A in t have an associated equivalence class, denoted by $eq(t, A)$. In a repair, a unique *target value* is assigned to each equivalence class E, denoted by $targ(E)$. BigDansing extends this algorithm to a distributed setting by modeling it as a distributed word counting algorithm. Unlike the common distributed word counting algorithm, BigDansing uses two map-reduce jobs. Consider as an example that there is a record, identified by an id, that has a violation that may be repaired by applying one of three fixes. Each one of these fixes will lead that record to a possibly new value (e.g., maintain the current value or assume a new value). In the first map-reduce job, the *map* tasks output a key-value pair of the form $([record\_id, new\_value], 1)$, which fed the *reduce* tasks. A *reduce* task receives all pairs with the same key value, and aggregates them by producing a new key-value pair of the form $([record\_id, new\_value], count)$, where $count$ is the number of pairs with the same key value. Then, the a second map-reduce job is scheduled, receiving as input the output of the previous *reduce* tasks. The *map* tasks produce a new key-value pair of the form $(record\_id, [new\_value, count])$ and deliver them to a set of *reduce* tasks. A record (identified by the $record\_id$ value) assumes the value that has the highest frequency (i.e., higher $count$ value).

3.3.4 RHEEM

RHEEM [1] is a cross-platform data processing tool. It has a different purpose from the other research data cleaning tools presented in this section. In fact, it does not execute a data cleaning program itself. Instead, it uses external platforms to perform the transformations that compose a data cleaning program, i.e., it plays the role of a middleware between applications and platforms.

RHEEM represents a data cleaning program as a graph composed by data transformations. For each data transformation, RHEEM chooses the best platform to perform it, cost-wise (e.g., the platform that has the best execution time or the less monetary-cost). However, choosing the best platform for each data transformation may result in a suboptimal data cleaning program, as the cost to perform the transformations and move data between platforms may be higher than running the transformations in a single platform. To create the most efficient data cleaning program, RHEEM has a cost-based cross-platform optimizer that: *(i)* is able to deal with the intricacies of each data cleaning platform, taking that burden from the users, *(ii)* takes data movement into account, and *(iii)* is able to deal with bad cardinality estimates, re-optimizing the execution of a data cleaning program while it is already executing.

To create a data cleaning program, the user needs to define a *RHEEM plan*, using Java, Python, the data-flow language proposed by the authors called *RheemLatin*, or the visual integrated development environment, *Rheem Studio*. A *RHEEM plan* is a directed graph whose nodes, commonly known as *operators*, represent the data transformations. Operators are connected by edges that represent the data flows between them. An operator is platform agnostic and abstracts a transformation. Examples of

RHEEM operators are the *Map* and *Reduce* operators, whose behaviour resembles those of the MapReduce paradigm, and the *GroupBy* operator, which groups records by a given key value. Each operator receives as input a data quanta. A *data quanta* is the smallest processing unit from the input datasets, independently of their format (e.g., in a RDBMS each record is a data quanta, in a document-store, the data quanta is a document).

Once a user defines a RHEEM plan, the cost-based optimizer receives it as input and produces an *execution plan*. The RHEEM plan defines the operations to be performed and their order, whereas the execution plan defines where those operations will be performed, i.e., in which platform (e.g., Spark, JavaStreams, PostgreSQL, etc). The optimizer will find the plan that minimizes the cost of the whole data cleaning program. Which platforms are selected depends on how the user defined the meaning of cost (e.g., monetary, execution time, minimize data movements, etc). In the remaining of this section, we detail the cost-based optimizer, also known as cross-platform optimizer.

The *Cross-Platform Optimizer* is responsible to select the most efficient platform to execute a single operator in a RHEEM plan. This optimizer does not perform any logical or physical optimizations, such as operator reordering, partitioning, etc. It is up to the user using RHEEM to optimize the plan, and to each platform to optimize an operator. The optimizer is divided into four phases: *(i)* plan inflation, *(ii)* cost estimates annotation, *(iii)* data movement planning, and *(iv)* plan enumeration.

Plan inflation is responsible for mapping RHEEM operators to platform-specific operators (e.g., Spark, JavaStreams). In this phase, RHEEM does not choose the platform where an operator will be executed. Instead, it maps an operator to every available platform that can execute it (i.e., creates an execution plan for an operator), and saves all generated mappings. The inflated plan of a RHEEM operator (gray box in Figure 3.13) is the group composed by that operator and its mappings.

RHEEM uses an UDF-based approach to evaluate the cost of each execution operator (i.e., the cost of executing an operator on a platform) in order to perform the *cost estimate annotation*. The user must define how he intends to measure the cost (execution time, monetary cost, etc) via an UDF dedicated to that effect, known as *cost function*. The cost estimates in RHEEM are not a single value, but instead an interval with the respective confidence that RHEEM has in that interval (pink box in Figure 3.13).

To enable the optimization of an execution plan, RHEEM uses the output cardinalities of each data operator (i.e., the number of records the data operator outputs), which are produced in the cost estimates annotation phase. To compute the cardinality of an operator, each operator has a cardinality estimator that takes into account how the operator works (e.g., number of iterations) and the input cardinality. Note that the output cardinality of the Reduce operator in Figure 3.13 (second blue box) is the input cardinality of the Map operator (third blue box). While executing the data cleaning program, if RHEEM detects that the cardinalities were badly estimated, it pauses the DCP execution and reoptimizes the physical plan according to the current execution state.

Data movement between platforms must be taken into consideration when creating an execution plan, since the cost of moving data among platforms may be higher than the operations themselves, if not performed properly (e.g., always broadcasting all records when we just need a subset). There are several ways of performing data movement, for example, using a broadcast operator (i.e., all-to-all)

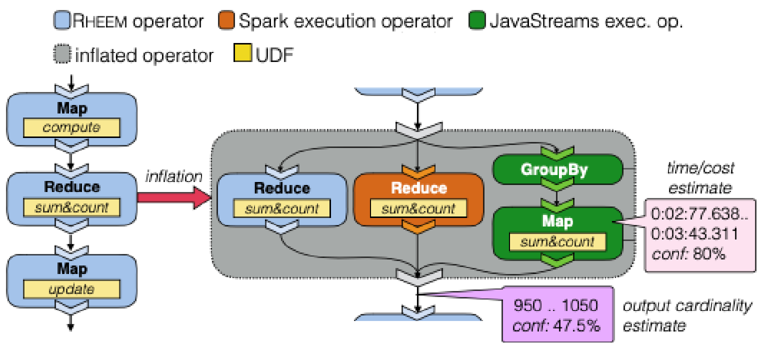


Figure 3.13: Inflated RHEEM operator (extracted from [1])

versus 1-to-n (i.e., one to *n* platforms, which is not necessarily all platforms) or n-to-m communication (i.e.,where *n* and *m* are subsets of the total platforms). Moreover, when communicating between platforms, we may need to apply transformations to the data being moved so that it can be used as input in the target platform. It is the job of the optimizer to find the best trade-off between data movement and transformation cost. RHEEM models the data movement paths across platforms as a graph problem, namely, as a channel conversion graph [25]. By handling the data movement across operators as a graph problem, RHEEM is able to find the most efficient strategy to perform the data movement (the algorithm is detailed in [25]). Once the most efficient strategy is found, it is attached to the execution plan of an operator.

After all the previous steps are completed (i.e., plan inflation, cost and cardinalities estimation, and data movement planning), the optimizer will identify which is the best execution plan for a given operator (*plan enumeration* phase). However, the optimizer does not select the best execution plan for a single operator. Instead, the optimizer considers the RHEEM plan as a whole, and tries to find the set of execution plans that minimize the overall cost. Without any optimizations, this would result in an exponential search space (with n operators and k execution plans for each, there would be k^n plans). To reduce the search space, RHEEM uses a lossless pruning technique [25], which guarantees that the optimizer is able to always find the optimal execution plan (i.e., the one with the minimum overall cost).

To execute and distribute the work between the platforms, there is the need to schedule that work in each one. The module responsible for this task is the *Executor*. Each platform receives a subplan of the whole execution plan, to that subplan we call it *stage*. A stage is a set of operators that: *(i)* are executed in the same platform, *(ii)* after finishing the work of that stage the control is returned to the executor, and *(iii)* the output of the stage is saved in a data structure instead of being pipeline to the following data operation (if it exists). The executor works as follows: it first dispatches all stages that do not have dependencies, and then dispatches the ones that have dependencies between each others (by only dispatching a stage after its dependencies are finished).

	CLEENEX	**CleanM**	**BigDansing**	**RHEEM**
Type	Transformation	Transformation	Rule	Cross-platform
Optimization Focus	DCP	DCP	DCP	Platforms
Logical Operators	Mapping, View, Matching, Clustering, Merging	FD, DEDUP, CLUSTERBY	Scope, Block, Iterate, Detect, GenFix	Map, Reduce, GroupBy
Physical Operators	Not explicitly defined	Not explicitly defined	PScope, PBlock, PIterate, PDetect, PGenFix, CoBlock, UCrossProduct, OCJoin	Depends on the platform used
Optimization Techniques	Inexistent	Single iteration k-means Operators coalescence	Operators coalescence Enhancers	On-the-fly plan reoptimization

Table 3.4: Summary of the data cleaning prototypes detailed in Section 3.3

3.3.5 Discussion

In this section, we presented four data cleaning prototypes whose design and implementation focus on scalability and distribution of a data cleaning program.

Table 3.4 summarizes the data cleaning tools presented (one column for each tool). The row *Type* indicates if the data cleaning tool is transformation-based, rule-based or cross-platform. The row *Optimization Focus* emphasizes the optimization focus of a tool, i.e., if it focus on the data cleaning program. The *Logical Operators* row defines which logical operators a tool offers. The row *Physical Operators* details the physical operators a tool defines. Finally, the *Optimization Techniques* row lists which automatic techniques a tool uses to enhance performance.

Most of the prototypes analyzed are transformation-based. This follows the approach of current commercial tools, which prefer the transformation-based approach. Apart from RHEEM, all tools try to optimize a data cleaning program (DCP) with the algorithms they provide. RHEEM is the exception since is the user's responsibility to optimize the plan (which represents a data cleaning program) as a whole and the platforms responsibility to optimize a subpart of a plan (e.g., if RHEEM decides that platform A should perform a subplan X, then the platform A is responsible to optimize that subplan X).

Regarding the logical operators, the two transformation-based data cleaning tools, CLEENEX and CleanM, have operations in common (e.g., mapping and DEDUP, clustering and CLUSTERBY). Although CLEENEX defines the Matching and View operators and CleanM does not, CleanM supports these operations through its declarative language. Regarding BigDansing, it differs from the transformation-based approaches, since it supports operations related to rule-based data cleaning tools, such as the detection of pair of tuples that do not satisfy a set of quality rules (Detect), and the repairing of the tuples that do not satisfy that set of quality rules (GenFix). Regarding RHEEM, it supports operators such as the Map, Reduce and GroupBy which resemble the mapping, merging and clustering operators of CLEENEX, respectively.

Apart from BigDansing, none of the platforms described in Section 3.3 explicitly define their physical operators. Instead, they define a set of algorithms that can execute a logical operator (e.g., in CleanM we can explicitly declare that we want to perform clustering with k-means). RHEEM physical operators are those of the platforms it uses.

Concerning the optimization techniques of each platform, CLEENEX currently performs the execution push down to the RDBMS and enables the user to give optimization hints. Both CleanM and BigDansing optimize their logical plans by performing operator coalescence (BigDansing performs this optimization in the plan consolidation phase). This optimization reduces the number of reads performed for a given dataset, which is a costly operation. RHEEM's most important optimization is the capability to re-adapt to bad statistics, by pausing execution and redefining the physical plan according to the current execution state.

Chapter 4

Proposed Solution

As stated in Section 1.3, the goal of this thesis is to implement an optimizer to be integrated in the data cleaning research prototype CLEENEX.

In Section 4.1 we describe the CLEENEX component architecture. Section 4.2 details the architecture of the optimizer and its components. There we introduce the notion of an execution plan, a physical operator, and a physical algorithm. In Section 4.3 we describe the cost model that supports the optimizer, namely how the output size of a physical algorithm is estimated, how the CPU and I/O cost are computed, and finally, how the cost model uses these measures to compute the cost of a physical algorithm. Finally, in Section 4.4 details the optimizations that were made in CLEENEX and that are not related to the optimizer.

4.1 CLEENEX Component Architecture

The CLEENEX component architecture is represented in Figure 4.1. It is composed of nine components, including the optimizer.

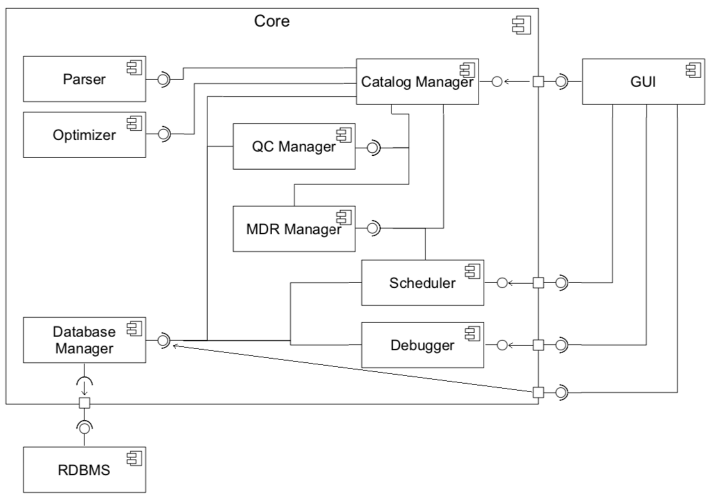


Figure 4.1: CLEENEX component architecture

A Data Cleaning Program (DCP) in CLEENEX is defined through a declarative language in the CLEENEX GUI. The DCP is firstly parsed by the Parser module. The Parser creates all necessary data structures to support CLEENEX execution. Among these structures are the Data Cleaning Graph (DCG) and the catalog. The catalog gathers all information about a DCP, being through it that the modules can access the DCG. The Catalog Manager (CM) acts as an intermediary between all modules, making available to all of them the catalog instance. The catalog instance is one of the dependencies of the renovated Optimizer. Note that the optimizer module already existed in CLEENEX. The difference to the new one is that it was not automatic, as it was dependent on the hints given by the user. Moreover, whereas the old had no output, the new one outputs an executable graph of transformations. This graph of transformations is delivered to the Scheduler so that the DCP declared in the CLEENEX GUI may be executed.

4.2 Optimizer Architecture

The design and implementation of an optimizer in CLEENEX is the main goal of this thesis. The optimizer is responsible for the following tasks: *(i)* receiving the DCG and translate it into an execution plan, an executable graph of transformations that lists which physical operators and physical algorithms should be used, *(ii)* generate equivalent execution plans, *(iii)* select the least costly execution plan, and *(iv)* save the cheapest execution plan for a given DCG so that it can be reused.

The DCG represents the logical plan in CLEENEX. In order to represent the physical plan, we designed another data structure, the Execution Plan (EP). Both DCG and EP represent a graph. However, whereas the DCG graph nodes are logical operators, the EP nodes are physical operators. A physical operator gathers all necessary information to execute a logical operator but does not execute it. Instead, it uses a physical algorithm. A physical algorithm is what enables the execution of a logical operator. Note that a physical operator may have several physical algorithms that are able to execute it. For example, the Matching physical operator has six physical algorithms.

The execution workflow of the optimizer is represented in Figure 4.2. The CLEENEX Executor represents the execution flow started by the HTTP request made by the user once he requests the DCP execution through CLEENEX GUI. Somewhere in the execution, the optimizer is requested to provide the cheapest execution plan for a given DCG. The optimizer starts by converting the DCG into an empty execution plan (Plan Converter). We consider an empty execution plan one that does not have any physical algorithm associated with its physical operators. Then, it checks if there is already an execution plan with the same characteristics in the Plan Cache module. If there is, then the cached execution plan is retrieved. Otherwise, the workflow proceeds, and from that empty execution plan, the Equivalent Plans Generator module creates, if possible, equivalent execution plans. It is expected that this module always outputs at least one execution plan, the one that uses the default physical algorithms of each physical operator. When a physical operator defines only one physical algorithm, that algorithm is considered the default for that physical operator. For the matching physical operator, the default physical algorithm is the Cartesian Product. Then, the cost of the plans is estimated by the Plan Cost Estimator module using a cost model. Once the cheapest plan is found, it is saved in the cache and is returned, and the CLEENEX Executor proceeds its execution.

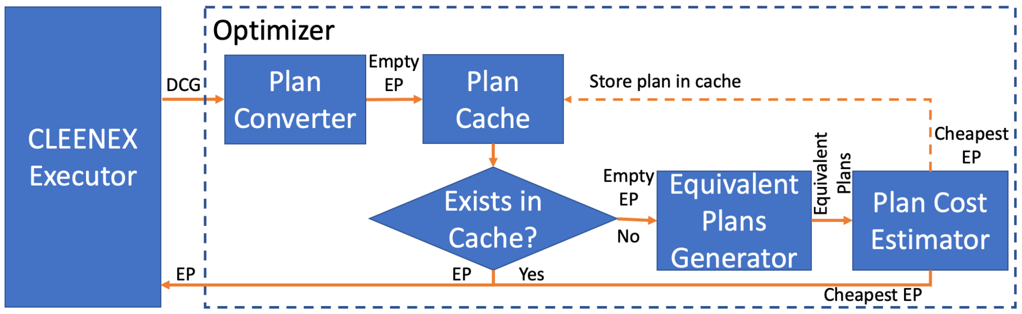


Figure 4.2: Optimizer execution workflow

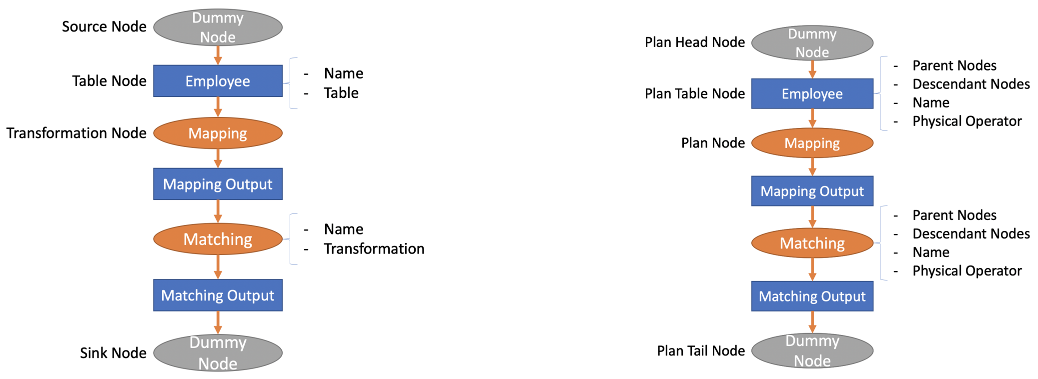
In the remainder of this section, we describe the several components that compose the optimizer. In Section 4.2.1 we describe what is the execution plan and how it compares to the data cleaning graph. In Section 4.2.2 we detail the algorithm that enables the optimizer to convert from a data cleaning graph to an execution plan. In Section 4.2.3 we explain how the cache module works, namely, how it saves an execution plan and how it evaluates if a given execution plan is equivalent to one already in cache. Finally, in Section 4.2.4 we detail how equivalent plans are generated.

4.2.1 Execution Plan

The execution plan has an identical structure to the DCG. It starts and finishes with two dummy nodes called *Plan Head Node* and *Plan Tail Node*, respectively. These nodes are needed for a matter of backwards compatibility since the structure that represents the DCG in CLEENEX contains this notion of head and tail in the graph. After the head node, we have a node for each input table, data transformation, and intermediary table. The data transformations are represented by a structure called *Plan Node* whereas the input tables and intermediary tables are represented by the *Plan Table Node*. The difference between these two structures is where they redirect the execution. The Plan Node is executed internally in CLEENEX. For its turn, the Plan Head Node is executed by an RDBMS. In Figure 4.3 the rectangular-shaped objects are Plan Table Nodes and the oval-shaped objects Plan Nodes. The number of nodes and their order is identical between the DCG and the EP. Figure 4.3 shows the translation of a DCG (Figure 4.3a) into an execution plan (Figure 4.3b), allowing us to see that they only differ in the information each node makes available, which in the DCG is more limited.

In the DCG, a table node contains information regarding the node name (*Name*), e.g., Employee or Mapping Output, and an instance that represents the table (*Table*). The transformation node contains also the node name (Name) and an instance that represents the transformation (Transformation). For its turn, the EP has the same data for both table and transformation node, containing a link to the descendant (Descendant Nodes) and parent nodes (Parent nodes), the node name (Name), and the physical operator that will enable the execution of a node (Physical Operator).

The plan nodes architecture is represented in Figure 4.4. The plan node, which represents a trans-



(a) Data Cleaning Graph Example

(b) Execution Plan Example

Figure 4.3: Example of a translation from a DCG to an execution plan

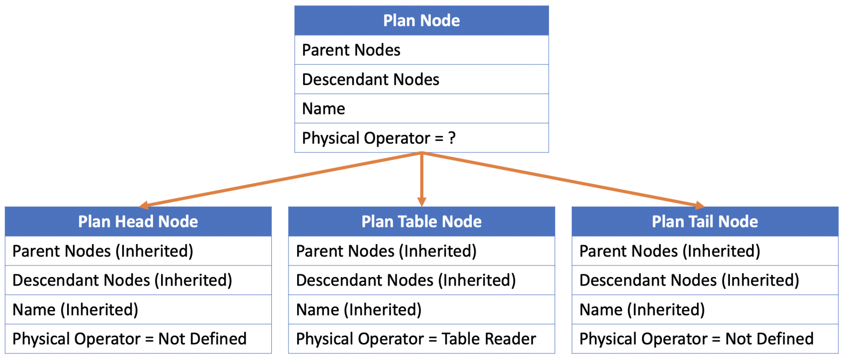


Figure 4.4: Architecture of the Plan Nodes

formation node, defines the necessary information that every node should have as well as the common behavior to all plan nodes, i.e., transformation nodes, table nodes, and dummy nudes. In other words, the Plan Table Node, the Plan Head Node, and the Plan Tail Node inherit the information to be stored and their behavior from the Plan Node. Note that only the transformation nodes (plan node) do not define a physical operator. This happens because this node, contrary to the others, does not have a fixed physical operator. The dummy nodes do not need a physical operator since they do not execute anything.

Each data transformation, such as matching, clustering, etc, has a physical operator. A Physical Operator contains the name of the node where it belongs which is inherited from the plan node where it is inserted, the list of supported physical algorithms, and the physical algorithm that is selected to execute that physical operator, also referred as the execution algorithm. This execution algorithm must be one of the physical algorithms listed under the list of the supported physical algorithms.

4.2.2 Plan Converter

The Plan Converter is responsible for translating a DCG into an EP. Listing 4.1 contains the pseudo-code for the conversion algorithm. It starts by creating a new instance of an Execution Plan, the empty execution plan (line 2). By default, this EP has two nodes, the Plan Head and Tail nodes. A map is also created as an auxiliary data structure to the conversion process, storing the translation between a DCG node and a plan node (e.g., the DCG's Source Node is translated into the EP's Plan Head Node). The conversion algorithm iterates through the descendants of a given DCG node (lines 15-25), translating each one into an EP node (lines 20-22). To support the iteration throughout the DCG nodes, a queue is used. This queue helps to avoid the more memory consuming recursive approach to perform the depth-first search in the DCG. Note that before converting a node, it is verified if the conversion didn't happen in a different iteration of the algorithm by querying the map for that node (lines 15-16). If the node is on the map, then it was already evaluated, otherwise, it needs to be converted.

```
 1 ExecutionPlan convertDCGToExecutionPlan(DataCleaningGrapg dataCleaningGraph) {
 2   ExecutionPlan executionPlan = new ExecutionPlan();
 3   // Keep track of the nodes already mapped
 4   Map nodesCreated;
 5   nodesCreated.put(dataCleaningGraph.getSource(), executionPlan.getHead());
 6   nodesCreated.put(dataCleaningGraph.getSink(), executionPlan.getTail());
 7   Queue queue;
 8   queue.add(dataCleaningGraph.getSource());
 9   // Represents a node in the DCG (logical plan)
10   Node currentLogicalNode;
11   while ((currentLogicalNode = queue.poll()) != null) {
12     PlanNode planNode = nodesCreated.get(currentLogicalNode);
13     if (currentLogicalNode.descendants().hasNext()) {
14       foreach descendantLogicalNode in currentLogicalNode.descendants()) {
15         PlanNode planNodeInMap = nodesCreated.get(descendantLogicalNode);
16         if (planNodeInMap == null) {
17           // Add this node to the queue so that its descendants can be verified
18           queue.add(descendantLogicalNode);
19           // Map Logical to Physical Node
20           planNodeInMap = PlanNodeFactory.build(descendantLogicalNode.getLogicalOperator());
21           // Add to the hashmap of mapped nodes to avoid redundant copies
22           nodesCreated.put(descendantLogicalNode, planNodeInMap);
23         }
24         executionPlan.addDescendantToNode(planNode, planNodeInMap);
25       }
26     }
27   }
28   return executionPlan;
29 }
```

Listing 4.1: Conversion Algorithm from Data Cleaning Graph to Execution Plan

The conversion from a DCG node to an EP node is delegated to a Plan Node Factory, as seen in Listing 4.1 at line 20. This factory decides which type of node should be created, if one representing a data transformation (e.g., matching, mapping, etc.), or a Plan Table Node representing either an input table or an intermediary table, i.e., the output of a data transformation. Moreover, this factory also extracts some information from the logical operator declaration that is needed in the physical operator. The View physical operator needs the query declared in the operator so that it can estimate its cost. The Matching operator also needs information concerning the window size, being that extracted from the declared hints in the logical operator.

4.2.3 Plan Cache

Caching the cheapest plan is essential to avoid having to repeat unnecessary operations, thus wasting time.

The optimizer is able to use any caching strategy since the strategy itself is abstracted by a Java interface called *Plan Cache*. Currently, there is only one caching strategy, the default one. The default strategy has limited storage of 100 plans. This limit was imposed to decrease CLEENEX's memory footprint. This caching strategy, for a given data cleaning program, stores the corresponding cheapest execution plan in terms of the overall plan cost.

When the optimizer requests an execution plan from the cache, it needs to send the empty execution plan generated when translating the data cleaning graph into an execution plan. An empty execution plan contains information regarding the data operations that will be performed and their order, the name of the input table, and its input size. The cache uses this information to compare that empty execution plan with every execution plan in cache, and it considers that there is a match if and only if: *(i)* the plans have the same nodes, i.e., have the same node names and same data transformations, *(ii)* all nodes are in the same order, and *(iii)* the input tables are equivalent. An input table is considered equivalent if: *(i)* the name of the input is the same, and *(ii)* the input size difference follows the rule $0.5 \times plan\_in\_cache\_input\_size \leq new\_input\_size \leq 1.5 \times plan\_in\_cache\_input\_size$. For example, assume that the cache finds an execution plan that is identical to the empty execution plan under test. If the cache plan has an input size of 1,000 records, for the plan under testing to be considered equivalent, it needs to have between 500 and 2000 records, i.e., half or double the cache plan input size, respectively.

4.2.4 Equivalent Plans Generator

The *Equivalent Plans Generator* (EPG) output is dependent on the physical algorithms that each physical operator has. The EPG uses the physical algorithms of each physical operator to create exhaustively, i.e., without any pruning technique, multiple equivalent execution plans. We consider execution plans as equivalent if: *(i)* the plans have the same nodes, i.e., have the same node names and the same physical operators, and *(ii)* all nodes are in the same order. The EPG generates all possible combinations of physical algorithms across the physical operators. If the user declares a DCP composed by logical operators whose corresponding physical operators only have one physical algorithm each (e.g., View, Clustering, Table, Mapping), the EPG only generates one execution plan. For example, a DCP with View and Mapping operators will generate only one execution plan, as can be seen in Figure 4.5. That execution plan uses the default physical algorithms. For a DCP that uses data transformations that have multiple physical algorithms, such as the Matching operator, then more than one execution plan will be generated. For example, for a DCP View and Matching operators, six equivalent execution plans are generated. Some of these plans that include are shown in Figure 4.6.

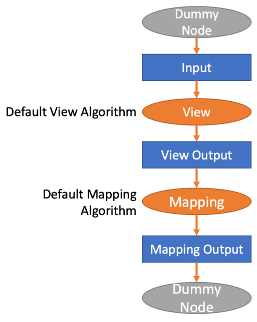
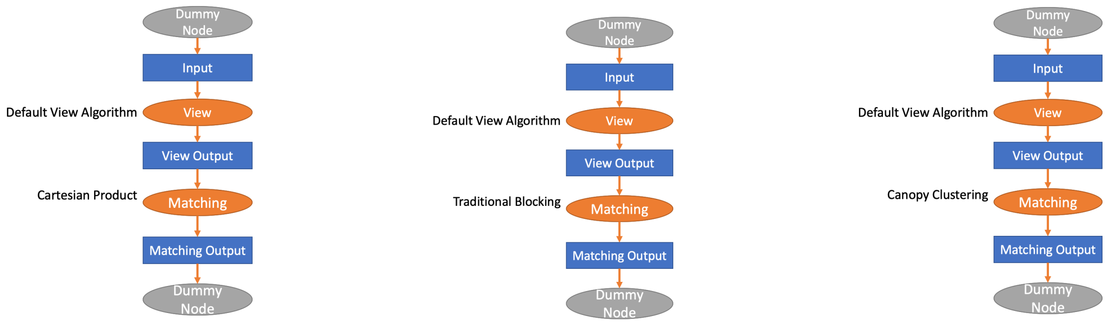


Figure 4.5: Execution plan for a DCP with a View and Mapping operator



(a) Plan with Cartesian Product (b) Plan with Traditional Blocking (c) Plan with Canopy Clustering

Figure 4.6: Example of some equivalent execution plans generated for a DCP with a View and Matching operator

The Equivalent Plans Generator pseudo-code is shown in Listing 4.2. The entry point is a method called $generateEquivalentExecutionPlans$ (line 1) that receives the empty execution plan created by the parser ($baseExecutionPlan$). This algorithm performs a Dept-First Search (DFS) to find all nodes in the empty execution plan (lines 6-27). As in the Plan Converter algorithm, explained in Section 4.2.2, a queue data structure is used to avoid a recursive DFS, which has a bigger memory footprint. The iteration starts by fetching the physical operator associated with the plan node (line 7) and its supported physical algorithms (line 8). The empty execution plan physical operators are always set with the physical operator first supported algorithm (line 11), i.e., the physical operator default physical algorithm. If the physical operator has more than one supported physical algorithm, then for each physical algorithm a new equivalent execution plan will be created (lines 23-25). In this case, the created execution plans will be identical to the $baseExecutionPlan$, i.e., the empty execution plan, differing only in the algorithm that the physical operator fetched at line 7 has as its execution algorithm. After that, all descendants of the node that is under evaluation are added to the queue so that they can be also evaluated.

The algorithm between lines 31 and 46, $createEquivalentPlans$ enables the generation of equivalent execution plans. It receives as input the plan node whose physical operator has multiple physical algorithms ($currentNode$), the already generated execution plans ($equivalentPlans$), and the number of supported algorithms by the physical operator ($supportedAlgorithms$). For each remaining physical algorithm, i.e., excluding the first one (at index 0), it copies all already existing plans stored in $equivalentPlans$ (line 36). Then, iterates in a new loop the copied plans (lines 37-44) and, for each one, searches the physical operator that triggered the execution of this algorithm in the copied plan (lines 39-40), and sets as the physical operator execution algorithm the one at index i in its list of supported algorithms (lines 41-43). To better understand the purpose of $createEquivalentPlans$ algorithm, consider the following example: assume that there are two equivalent plans already generated, A and B. These plans have some nodes, and we want to add another node whose physical operator supported physical algorithms are X and Y. If there are two plans, the algorithm needs to create two versions of those two plans. The first version with plan A and B using physical algorithm X and another version using physical algorithm Y. Thus, from two equivalent plans, we go to four.

```
1  ExecutionPlan[] generateEquivalentExecutionPlans(ExecutionPlan baseExecutionPlan) {
2     ExecutionPlan [] equivalentPlans;
3     Queue nodesToEvaluate;
4     nodesToEvaluate.put(baseExecutionPlan.getHead().getDescendantNodes());
5     PlanNode currentNode = nodesToEvaluate.poll();
6     while(currentNode != null) {
7        PhysicalOperator physicalOperator = currentNode.getPhysicalOperator();
8        Algorithm[] supportedAlgorithms = physicalOperator.getSupportedAlgorithms();
9        // Affect the baseExecutionPlan (empty plan, index = 0)
10       // with the first supported algorithm
11       physicalOperator.setExecutionAlgorithm(supportedAlgorithms[0]);
12       // Update the other execution plans (if they exist) with the same algorithm
13       foreach ep in <remaining execution plans> {
14          // Get the node that is being manipulated in the other plan
15          PlanNode node = ep.searchPlanNodeInExecutionPlan(currentNode);
16          // Update the algorithm
17          node.getPhysicalOperator()
18          .setExecutionAlgorithm(
19             node.getPhysicalOperator().getSupportedAlgorithms()[0]
20          );
21       }
22       // When there is more than one supported algorithm create equivalent plans
23       if (supportedAlgorithms.length() > 1) {
24          createEquivalentPlans(currentNode, equivalentPlans, supportedAlgorithms.length());
25       }
26       nodesToEvaluate.addAll(currentNode.getDescendantNodes());
27    }
28    return equivalentPlansGenerated;
29 }
30
```

```
createEquivalentPlans(PlanNode currentNode, ExecutionPlan[] equivalentPlans,
int supportedAlgorithms) {
  // index = 0 was already used to affect the empty execution plan (baseExecutionPlan)
  from i = 1 to supportedAlgorithms {
    // Creates a copy of the plans that are in equivalentPlans.
    ExecutionPlan [] plansToCreate = copyAllPlans(equivalentPlans);
    foreach executionPlanCopy in plansToCreate {
      // Find the corresponding node in the new plan and set the algorithm
      PlanNode currentNodeInCopyPlan =
      executionPlanCopy.searchPlanNodeInExecutionPlan(currentNode);
      PhysicalOperator physicalOperator = currentNodeInCopyPlan.getPhysicalOperator();
      Algorithm algorithm = physicalOperator.getSupportedAlgorithms()[i]);
      physicalOperator.setExecutionAlgorithm(algorithm);
    }
    i++;
  }
  equivalentPlans.addAll(plansToCreate);
}
```

Listing 4.2: Equivalent Execution Plan Generator Pseudo-Algorithm

4.3 Cost Model

The *Cost Model* enables to measure the cost of an execution plan. First, we need to take into account that a plan is composed by a set of physical operators. In a given execution plan, each physical operator has only one execution algorithm. This execution algorithm is selected among the list of supported physical algorithms for that physical operator. The plan cost is affected by the cost of the physical algorithms that are executed, i.e., it is not possible to define the cost for a physical operator since it can be executed by different physical algorithms (e.g., the Matching can be executed by six different algorithms). Therefore, the cost model needs to be created on an physical algorithm-basis. This cost model includes three measures that are addressed individually in the remaining of this section: *(i)* the estimated output size *(ii)* the cost of a physical algorithm, which allows us to evaluate how much CPU effort is needed to execute the algorithm, and *(iii)* the I/O cost, measured in the number of pages read from disk, allowing to measure the I/O effort needed to retrieve all data required to perform the algorithm.

In Section 4.3.1 we detail how the output size of an operator is estimated. Section 4.3.2 describes the cost formulas used to compute both the CPU and I/O cost. Lastly, in Section 4.3.3, we explain how the final cost of a physical algorithm is computed.

4.3.1 Output Size Estimation

To be able to properly estimate the output size of each data operator, we need to perform an in-depth analysis of each operator semantics to understand how it works. CLEENEX has five logical and physical operators, view, merging, clustering, matching and mapping, as discussed in Section 3.3.1. Table 4.1 shows, for each physical algorithm, its estimated output size and the expected maximum output size. In Table 4.1, N_R refers to the number of records of a relation R, and N_S to the number of records of a relation S, $V(attr_R)$ is the number of distinct values for a given attribute $attr$ of relation R, and w is the window size in the Sorted Neighborhood physical algorithms. We use $N_{R \cup S}$ to refer the size of the union between relations R and S, and $BKV_R \cup BKV_S$ is the total number of blocking key values from both input relations R and S.

Physical Operator	**Physical Algorithm**	**Estimated Output Size**	**Expected Maximum Output Size**
View	-	N_R	?
Mapping	-	N_R	$\geq N_R$
Merging	Sorting	$V(attr_R)$	N_R
	Hashing		
Clustering	Transitive Closure	N_R	$2N_R$
Matching	Cartesian product	$N_R \times N_S$	$N_R \times N_S$
	Traditional Blocking	$\frac{N_R \times N_S}{\left(\sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i}\right)^2} \times \sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i^2}$ [6]	$N_R \times N_S$
	Canopy Clustering	$\frac{N_R \times N_S \times n_l^2}{n_t^2}$ [6]	$N_R \times N_S$
	Sorted Neighborhood Join (SNJ)	$(w^2 + 2(N_R + N_S - w)(w-1))$ [6]	$N_R \times N_S$
	Inverted Index SNJ	$\frac{N_R \times N_S}{V(BKV_R \cup BKV_S)^2}(w^2 + (V(BKV_R \cup BKV_S) - w)(2w-1))$ [6]	$N_R \times N_S$
	Adaptive SNJ	$\frac{N_R \times N_S}{\left(\sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i}\right)^2} \times \sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i^2}$	$N_R \times N_S$

Table 4.1: CLEENEX output size estimation. N_R stands for the number of records of relation R, whereas N_S, is the number of records of S. $V(attr_R)$ is the number of distinct values for attribute $attr$ of relation R, and w is the window size in the Sorted Neighborhood algorithms. We use $N_{R \cup S}$ to refer the size of the union between relations R and S, and $BKV_R \cup BKV_S$ is the total number of blocking key values from both input relations R and S

The view operator corresponds to a SQL query whose execution will be pushed into the RDBMS. It is not possible to estimate which will be its output size since no optimizer can predict which query the user will insert. However, after knowing the query associated with the view operator, the optimizer can request the RDBMS to estimate the output size and the record size (i.e., how many bytes each record has), if the queried table already exists. If this queried table does not exist, i.e., it is an intermediary table, then the output size is estimated to be the input size, N_R.

The mapping operator enables the users to map a given record into zero or more records (e.g., split a column into multiple columns). It is not possible to state that in all cases we will have the same output size as input size, i.e., N_R. However, the estimated output size of N_R is a compromise we assumed to enable the output estimation of this operator by the optimizer. Regarding the maximum output size, we expect it to be greater or equal to N_R given that the operator may output 0 or more output records for each input record.

The merging semantics of this operator enables the aggregation of records and latter collapsing them into once record using an aggregation function. This operator resembles the group by clause in SQL. After aggregating the records into several groups, it collapses them into a single record. The estimated output size is dependent on the number of distinct values that the grouping attributes have, i.e., $V(attr_R)$.

The maximum output size is N_R, the number of input records, occurring when all the groups generated have a size of 1. Regarding the physical algorithms that allow the execution of this operator, we have two possibilities: *(i)* merge-sort based algorithm, or *(ii)* hash-join based algorithm. The physical algorithm used does not affect either the estimated or maximum output.

The clustering operator enables the user to partition a given relation into several groups of records, each identified by a unique *cluster id*. It receives and outputs one relation. Each record of the input relation must represent a pair of records, i.e., the typical output of the matching operator. The records contained in the output relation have the schema $\{cluster\_id, record\_id\}$, where $cluster\_id$ identifies a cluster, and $record\_id$ a record that belongs to that cluster. The output relation may contain multiple records whose $cluster\_id$ is the same, meaning that a cluster has several records in it. The physical algorithm used to implement the clustering operator in CLEENEX is the transitive closure. The operator's output is estimated to be N. The reasoning behind this estimation is that there are pairs of records that repeat a given record multiple times. For that record, only one final record in the clustering output will be needed. For example, for records A, B, C, and D, and pairs $B-B$,$B-A$, $B-C$, and $B-D$, the following records will be output: *(i)* $cid = 1, B$, *(ii)* $cid = 1, A$, *(iii)* $cid = 1, C$, and *(iv)* $cid = 1, D$, with cid being the cluster identifier. That is, for four pairs, four records are output by the cluster operator, thus justifying an estimated output size of N. For the worst-case scenario, where there are not any identical records, it will output $2N_R$ records, since two records are created for each pair. Therefore, both the estimated output size and expected maximum output size are equal to $2N_R$. This happens since for each pair of records received as input, it outputs two records, the cluster to where each record is put does not affect the output size of the operator.

The matching operator enables to detect approximate duplicates. It receives two relations as input and outputs a single relation whose records have the schema $\{R\_record, S\_record, d_{RS}\}$, where d_{RS} is the distance between records R_record and S_record. The two input relations can be a single relation that is given as input twice. This operator has more than one alternative physical algorithm in CLEENEX. Each algorithm has a trade-off between performance and accuracy of the results, as discussed in 3.1.6. Unlike the estimated output size, the maximum output is independent of the algorithm, occurring when all records are duplicates, thus generating $N_R \times N_S$ records. To estimate the output size of each matching physical algorithm, with the exception of the Cartesian Product, we use the results reported in [6]. Those results have two variants: *(i)* with a normal distribution, which assumes an equal probability for any word to appear in a dataset, and *(ii)* with a Zipf distribution, that assumes that in a list of words ordered by their frequencies, the word at position p has a relative frequency of $1/p$. We use the normal distribution in for the estimations of all our physical algorithms except the Traditional Blocking and Adaptive SNJ. The reasoning behind this difference for the remaining matching physical algorithms was due to the lack of accuracy of the normal distribution for these two algorithms in specific. The authors estimate the number of candidate pairs that each matching physical algorithm will produce. [1] The survey used a uniform distribution for the blocking key values, meaning that every block will have roughly the same size. A blocking key represents a record based in one or more attribute values of that record. It takes

[1] This cannot be confused with the matching operator output, which already implies the filtering of non-similar pairs.

into account the characteristics of each physical algorithm to estimate its output. In the estimated output size formulas, $V(BKV_{R/S})$ represents the number of unique blocking key values in either relation R or S, whereas $V(BKV_R \cup BKV_S)$ is the number of distinct key values in the union between both relations, n_l and n_t are the canopy's thresholds that were introduced in 3.1.5, and w refers to the window size of a Sorted Neighborhood Join (SNJ) physical algorithm. Note that the estimation of the Adaptive SNJ is not as accurate as the others since it has a varying window size.

For some of these estimations, there is the need to know the number of distinct values for a given attribute, $V(attr_R)$. However, in the most common case, which is when we are estimating the output size of an intermediary data transformation, i.e., an operator that uses the output of another operator and not directly the input table of the DCG, in CLEENEX, there is no way to know how many distinct values there are. Note that the output of the data transformations do not exist when the optimizer is computing the estimations since they were not yet executed. As a workaround, the number of distinct values is extracted directly from the node input size, which in the common case is also an estimation. The number of distinct values is given as a multiplication of the input size by a distinct values factor. In the current implementation, this factor is 30%. There is no reasoning behind this value. Moreover, in the future it would be advisable to have another module that is able to perform the estimation of distinct values for any data transformation at any node in the graph.

4.3.2 CPU and I/O Cost

To measure the cost of a physical algorithm, as mentioned earlier, the cost model needs to evaluate both the algorithm's performance, or CPU cost, measured using the complexity (*Big O Notation*), and the impact in terms of I/O cost, commonly measured in terms of blocks/pages read from disk into memory. Table 4.2 shows, for each physical operator, and respective physical algorithm(s), its CPU cost and I/O cost. In what concerns the I/O Cost, we provide formulas for the expected cost in a RDBMS, that is the I/O cost of perming the operation in an RDBMS, and the expected cost having in mind how the operators are implemented in CLEENEX. In Table 4.2, N_R refers to the number of records of a relation R, b_R to the number of blocks needed to store all the records from relation R, and w to the window size in the Sorted Neighborhood algorithms. Finally, M is the number of pages the memory can accommodate. When there is a second relation, i.e., in the matching operator, we refer to another relation S. To refer the size of the union between relations R and S we use $N_{R \cup S}$.

The Mapping physical operator reads all records from the input table and, for each one, performs an action. It is not possible to accurately predict the algorithmic cost since it is dependent on the User Defined Functions (UDFs) CPU cost, which for CLEENEX, and consequently, its optimizer, is a black box. Therefore, we used N_R as the most probable algorithmic cost for the operator. Regarding the I/O cost, the mapping default physical algorithm reads all the contents from the input table, thus b_R, then it inserts the results of the mapping in another table. This insertion, due to the nature of the mapping operator, can either be zero records, one for each input record, or multiple records for each input one. As a compromise, we assumed we would write, on average, as many records as the input ones. That is,

Physical Operator	Physical Algorithm	CPU Cost (Big O Notation)	I/O Cost in RDBMS (Pages)	I/O Cost in CLEENEX (Pages)
Mapping	-	$O(N_R)$	$2b_R$	$2b_R$
Merging	Sorting	$O(N_R log(N_R))$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$
	Hashing	$O(N_R)$	$3(b_R) + 4N_R$	$3(b_R) + 4N_R$
Clustering	Transitive Closure	$O(N_R)$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 8) + 4N_R$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 8) + 4N_R$
Matching	Cartesian Product	$O(N^2_{R\cup S})$	$b_R * b_S + b_R$	$b_R * b_S + b_R$
	Traditional Blocking	$O(N^2_{R\cup S} \times log(N_{R\cup S}))$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 2 + b_S)$	$b_R + b_S$
	Canopy Clustering	$O(N^2_{R\cup S} \times log(N_{R\cup S}))$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 2 + b_S)$	$b_R + b_S$
	Sorted Neighborhood Join (SNJ)	$O(N_{R\cup S} \times log(N_{R\cup S}) + w)$	$b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 3 + b_S) + b_S$	$b_R + b_S$
	Inverted Index SNJ	$O(N_{R\cup S} + log(N_{R\cup S}) \times w)$	$(b_R + b_S)(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 6)$	$b_R + b_S$
	Adaptive SNJ	$O(N_{R\cup S} \times log(N_{R\cup S}) + w)$	$2(b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)) + b_R \times b_S + b_R$	$b_R + b_S$

Table 4.2: CPU and I/O cost analysis of CLEENEX physical operators. N_R refers to the number of records of a relation R, b_R to the number of blocks needed to store all the records from relation R, and w to the window size in the Sorted Neighborhood algorithms. Finally, M is the number of pages the memory can store. When there is a second relation, i.e., in the matching operator, we refer to another relation S. To refer the size of the union between relations R and S we use $N_{R\cup S}$

for N records we would output, on average, N records. Therefore, we will write b_R pages. This makes the mapping I/O cost equal to $2b_R$.

It is possible to have two merging physical algorithms, sorting and hashing. Each one has a different CPU and I/O cost. However, CLEENEX only implements one of the algorithms, the hashing. Even though CLEENEX has not yet implemented the sorting algorithm, we developed its cost model to future-proof the developed cost model. The sorting algorithm is typically a variant of a Merge-Sort whose complexity is known to be $O(N_R log(N_R))$. In what concerns the I/O cost, it is also known to need $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$ pages. The hashing algorithm puts each input record into a given bucket, thus having a complexity of $O(N_R)$. The I/O cost of a hashing algorithm is $3(b_R) + 4N_R$, the same as in an RDBMS.

The transitive closure implemented in CLEENEX performs the following operations: *(i)* projection of each input record $Dataset_R : T\_record, V\_record$, into two records, one for each sub-record in the input, i.e., $Dataset_A : T\_record; Dataset_B : V\_record$, *(ii)* create a unified dataset with the union of the previously created datasets, *(iii)* iterate through that dataset, and for each record, create a new one with the schema $cluster\_id, record\_id$, wherein a first phase, the $cluster\_id$ is unique for each record, *(iv)* iterate $Dataset_R$ again, and apply the transitive closure, updating the cluster identifiers of those records that should be in the same cluster. Summarizing, the algorithm iterates twice over the input dataset, thus having a complexity of $O(2N_R)$, which simplifies to $O(N_R)$, and demanding two page reads ($2b_R$). Moreover, before the algorithm iterates over the dataset, it needs to perform two projections, whose cost in total is $2b_R$. To perform the union, assuming the hashing join is used, it has a page cost of $3b_R + 4N_R$. Finally, the dataset needs to be sorted, thus summing to the previous I/O costs, the external merge-sort one, i.e., $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$. All of these costs sum up to $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 8) + 4N_R$.

The Matching physical operator has a total of six physical algorithms.The Cartesian Product is a nested-loop join, therefore assuming its CPU Cost, $O(N_R^2)$, and I/O cost, $b_R \times b_S + b_R$. These algorithms use the RDBMS only as a provider of the input data. Every physical algorithm performs the same action before it starts executing: reads all the input data from the RDBMS and saves it in an in-memory data structure, thus explaining the $b_R + b_S$ I/O cost, i.e., the cost of reading all pages from both input datasets.

We now address the expected RDBMS cost for each matching physical algorithm. The RDBMS cost would be the estimated I/O cost if CLEENEX implemented distributed physical algorithms. Distributed algorithms have the same assumption as an RDBMS: it may not be possible to have all input data in memory at the same time. This is why we analyzed each physical algorithm RDBMS cost.

The Traditional Blocking first performs an aggregation, which can be performed through hashing or sorting. Since we cannot know which one will be used by an RDBMS we assumed that the sorting was used, thus having a cost of $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$. After aggregating all records, the Cartesian product is performed in each group, having a cost of $b_R \times b_S + b_R$. Thus, the Traditional Blocking has a total cost of $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 2 + b_S)$.

Regarding the Sorted Neighborhood Join (SNJ) and Adaptive SNJ, they need to divide their execution into two phases: *(i)* sorting, and *(ii)* comparison. For sorting the external sort-merge is used, thus assuming its cost, $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$. For the comparison phase, a Cartesian product is performed inside each group of records, thus having a cost of $b_R \times b_S + b_R$. Moreover, before sorting, there is also the need to merge both relations into a single one, i.e., a union, so that the sorting can be performed. Assuming the union is performed by a Merge-Join algorithm, it has a cost of $b_R + b_S$. This sums up to a cost of $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 3 + b_S) + b_S$.

The Canopy Clustering and Traditional Blocking physical algorithms are very similar. The difference resides only in the way the groups are created and how the blocking key is used. Remember that the blocking key represents a record in the Matching physical operator. It is composed by one or more attributes of a record. Both physical algorithms read the input data in the same way, thus having the same I/O cost.

The Inverted Index SNJ will first get all distinct values, which has the same cost as an aggregation. As in the other operators, we assume that the sorting algorithm is used, thus having a cost of $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$. In a second step, the blocking keys are sorted. Assuming the external sort-merge is used, it comports a cost of $b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)$. Finally, the Cartesian product is performed inside each window. These costs sum up to $2(b_R(2\lceil log_{M-1}(\frac{b_R}{M})\rceil + 1)) + b_R \times b_S + b_R$.

All matching physical algorithms have to iterate through the input datasets, thus having a CPU cost of $O(N_{R\cup S})$. However, since they all have this factor in common, it is not considered in the cost model. The Cartesian product CPU cost is known to be $O(N^2_{R\cup S})$. In the remainder of this section, we analyze the CPU cost for the remaining physical algorithms of the Matching physical operator.

The Traditional Blocking physical algorithm creates several groups and performs the Cartesian product ($O(N^2_{R\cup S})$) in each one. It enables spreading the quadratic operation throughout the various groups, having a final complexity of $O(N^2_{R\cup S} \times log(N_{R\cup S}))$.

The Canopy Clustering, after reading all records, iterates throughout them all ($O(N_{R\cup S)})$ and generates a token for each one. Then, it iterates through those tokens and creates an inverted index that uses to group the records. This inverted index has the mapping between a token and its records. Inside each group, the Cartesian product is performed. Assuming that these tokens allow a similar division as in Traditional Blocking, $log(N_{R\cup S})$ groups are created. All of these costs sum up to a total of $O(N_{R\cup S} + N^2_{R\cup S} \times log(N_{R\cup S}))$, which can be simplified to $O(N^2_{R\cup S} \times log(N_{R\cup S}))$, since $N_{R\cup S}$ is not the dominant parcel, since there is a $O(N^2_{R\cup S}$ parcel.

The Sorted Neighborhood Join algorithms have all a similar cost. The difference between the SNJ and Adaptive SNJ is that in the latter we cannot use the window size as a variable in the cost model since it is not of a fixed size. Both need to sort their input. They both use Java's Collection sorting algorithm [2] that has a complexity of $O(N_{R \cup S} \times log(N_{R \cup S}))$. The two algorithms then proceed to iterate throughout the sorted records, $O(N_{R \cup S})$, and create the record pairs. The difference between the SNJ and Adaptive SNJ is that the first multiplies the known window size by the number of iterated records, whereas the latter cannot since it is unknown prior to the algorithm's execution.

The Inverted Index SNJ starts by iterating all records to create an inverted index data structure with all distinct blocking key values, $O(N_{R \cup S})$. Then, using a fixed window size, it iterates throughout those distinct blocking key values and creates the candidate record pairs. This iteration is estimated to have a complexity of $O(log(N_{R \cup S} \times w)$. Therefore, the Inverted Index SNJ CPU cost is $O(N_{R \cup S}) + log(N_{R \cup S}) \times w)$.

The view physical algorithm and the table reader algorithm share the same cost model. There is no CPU cost, i.e., $O(0)$, since the operation is done all in the RDBMS. Regarding the I/O cost, it can be measured in two possible ways. If the query can be executed when the optimizer is estimating the cost, which occurs when the table used by the query already exists, i.e., the input of the operator is not the output of another operator, then the I/O cost is retrieved from the database using a SQL function available in PostgresSQL, the *EXPLAIN* function. The EXPLAIN function retrieves among other information, the query's estimated output size and the size, in bytes, of each record. The I/O cost is the multiplication of those two values. However, if the query cannot be executed when estimating the cost, the optimizer assumes that both the input size and the size of each record is given by the parent node, having as the operator's I/O cost the multiplication of those two values.

4.3.3 Cost of a Physical Algorithm

The cost of a physical algorithm is given by the sum of its estimated I/O and CPU cost. The current cost model allows defining a weight for each parcel, i.e., we can give a different weight to the I/O and CPU cost in the computation of the algorithm's final cost. At the moment both costs have an equal weight, i.e., each one contributes 50%.

The cost of each non-default matching physical algorithm is influenced by the quality of its results (effectiveness) and its performance when compared against the matching default physical algorithm, the Cartesian product. For example, the Sorted Neighborhood Join will not detect as many duplicates as the Cartesian Product, but will also not take as much time to complete. The cost model takes this into account in what we called the penalization factor. The sum of the CPU and I/O cost is multiplied by this factor, i.e., $(CPU\ Cost + I/O\ Cost) \times Penalization\ Factor$. The penalization factor is a percentage that can vary between 50% and 200%. That is, at maximum, it either cuts by half the original physical algorithm cost or duplicates it. The purpose of limiting these values is to guarantee that no physical algorithm has such a big bonus or penalization, that is either always chosen or never chosen. We consider a bonus if the penalization factor is below 100%, since it decreases the total cost, thus increasing

[2]https://docs.oracle.com/javase/8/docs/api/java/util/List.html#sort-java.util.Comparator-

the chances of that physical algorithm to be chosen by the optimizer. In contrast, a penalization factor above 100% is a penalization since it increases the final physical algorithm cost.

The penalization factor and cost formula for each physical algorithm is shown in Table 4.3. The *Final Penalization Factor* column shows the physical algorithm penalization factor. The penalization factor of a physical algorithm is the average between two factors: *(i)* the output factor, that takes into account the algorithm effectiveness, and *(ii)* the performance factor, that evaluates the performance gain achieved by using a given physical algorithm when compared to the default physical operator algorithm. Finally, the column *Cost Formula* shows the final cost formula that takes into account the CPU cost, I/O cost and the penalization factor.

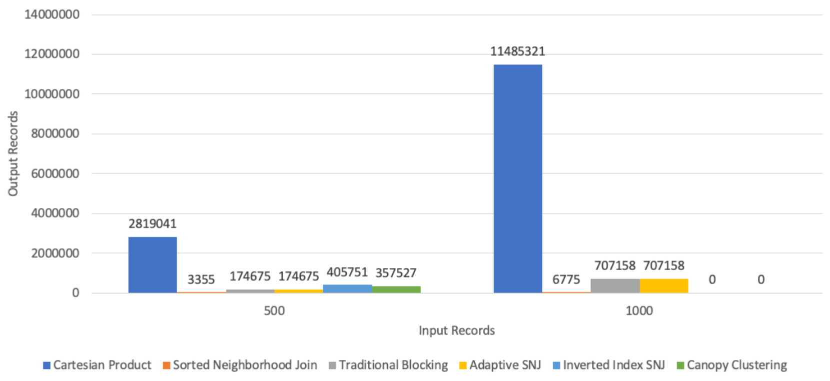
Physical Operator	Physical Algorithm	Output Factor	Performance Factor	Penalization Factor	Cost Formula
View	Default	-	-	100%	$(CPU + I/O) \times 100\%$
Mapping	Default	-	-	100%	$(CPU + I/O) \times 100\%$
Merging	Default	-	-	100%	$(CPU + I/O) \times 100\%$
Clustering	Default	-	-	100%	$(CPU + I/O) \times 100\%$
Matching	Cartesian Product	-	-	100%	$(CPU + I/O) \times 100\%$
	Sorted Neighborhood Join	199.91%	0.01% → 50.00%	124.96%	$(CPU + I/O) \times 124.96\%$
	Traditional Blocking	193.82%	0.35% → 50.00%	121.91%	$(CPU + I/O) \times 121.91\%$
	Adaptive SNJ	193.82%	8.47% → 50.00%	121.91%	$(CPU + I/O) \times 121.91\%$
	Inverted Index SNJ	185.61%	5,013.83 → 200.00%	192.81%	$(CPU + I/O) \times 192.81\%$
	Canopy Clustering	187.32%	1,619.64% → 200.00%	193.66%	$(CPU + I/O) \times 193.66\%$

Table 4.3: Physical algorithm's cost formulas with penalization factor

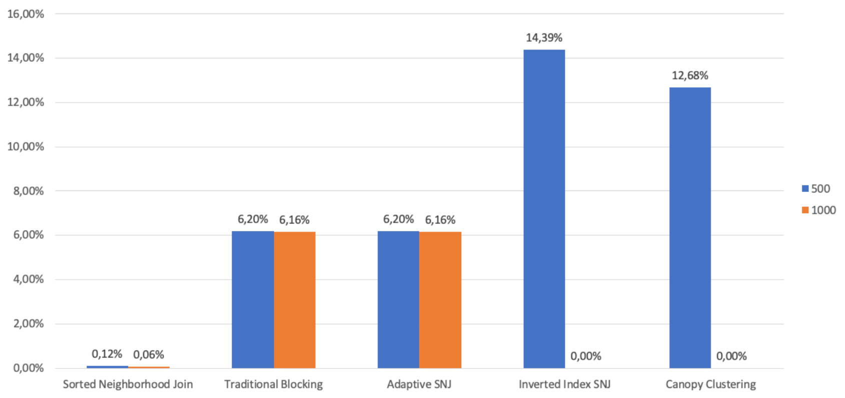
To compute the output factor of each physical algorithm, we studied how each one performed against the default physical operator algorithm. Since only the matching physical operator has more than one supported physical algorithm, the study focused on that physical operator. The study, shown in Figure 4.7, compares the output of each physical algorithm, i.e., the number of candidate record pairs generated, with input sizes of 500 and 1,000 input records. Figure 4.7a shows the output of each physical algorithm for both input sizes.

The Cartesian product generates all possible candidate record pairs. All records are compared against each other, guaranteeing that all possible duplicates are detected, i.e., an effectiveness of 100%. However, it also creates a big difference in the output size, i.e., the number of candidate record pairs generated, when compared against the remaining matching physical algorithms. Figure 4.7b makes a direct comparison between the non-default matching physical algorithms and the Cartesian product's output. The purpose of this comparison is to evaluate the physical algorithms' effectiveness since it is correlated with the number of candidate record pairs generated. For example, if a physical algorithm generates more record pairs than another, then its effectiveness is expected to be better since it will compare more records, thus having more opportunities to detect duplicates.

The output factor is always a penalization since no physical algorithm outputs more candidate record pairs than the Cartesian Product. To obtain the output factor we subtract 200%, the maximum penalization factor, to the average percentage of candidate record pairs generated in comparison with the Cartesian Product, shown in Figure 4.7. For example, the Sorted Neighborhood Join with 500 input records, generated 0.12% of Cartesian's output, and with 1,000 input records, only 0.06%. The average SNJ output, when compared against the Cartesian product, is 0.09%. Therefore, the SNJ's output factor is 200% - 0.09%, that is, 199.91%.



(a) Matching physical algorithms output sizes- number of candidate record pairs generated

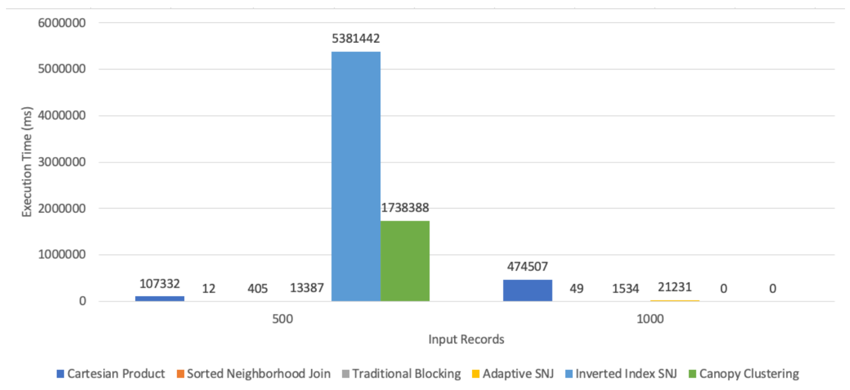


(b) Efficiency of the non-default matching physical algorithms when compared against the Cartesian product

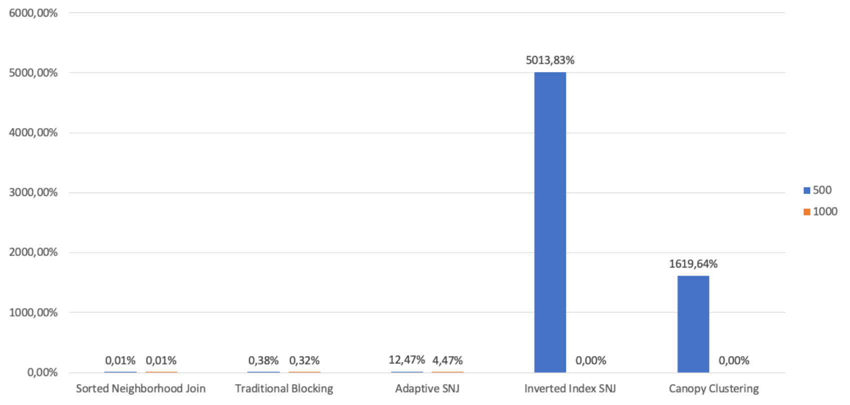
Figure 4.7: Time comparison between matching physical algorithms

Regarding the performance factor, we performed the same study as for the output factor but we measured the performance instead. The performance is measured in terms of execution time. The performance study is shown in Figure 4.8. In Figure 4.8a, it is possible to see how much time, in milliseconds (ms), each physical algorithm took to generate their candidate record pairs for both 500 and 1,000 input records. Surprisingly, both Canopy Clustering and Inverted Index SNJ are worst performers than the Cartesian product. For its turn, the Sorted Neighborhood Join is the best performer, which contrasts with the results shown in Figure 4.7b, where it is the less efficient physical algorithm. Figure 4.8b makes a direct comparison between the non-default matching physical algorithms and the Cartesian product's performance. This comparison allows us to evaluate how much time we gained by having the optimizer to choose a non-default matching physical algorithm. For example, if the optimizer uses the Inverted Index SNJ, then the execution time will be 5,013.83% greater than if it had used the Cartesian product physical algorithm.

The performance factor is not always a penalization since most algorithms are better performers than the Cartesian Product. To compute the performance factor we used the average performance when compared to the Cartesian Product, shown in Figure 4.8. That average performance is used directly as the performance factor, being only caped by the lower and upper penalization factor limits of 50% and 200%, respectively. This means that for a physical algorithm such as Traditional Blocking,



(a) Matching physical algorithms performance - execution time (ms)



(b) Comparison between non-default matching physical algorithms performance and Cartesian Product performance

Figure 4.8: Performance comparison between matching physical algorithms

whose execution time, on average, is 0.35% of the Cartesian Product, the final performance factor is 50%. On the opposite side, for algorithms such as Inverted Index SNJ, that executes, on average, in 5,013.85% of the Cartesian Product's execution time, the performance factor is capped to 200%.

To obtain the estimated cost of an execution plan, we need to sum the cost of each of its physical algorithms. For example, consider the execution plan represented in Figure 4.9 which has three physical operators, and consequently, three selected physical algorithms. According to Table 4.3, traditional blocking has a penalization factor of 121.91%. This means that its cost will be increased by a factor of 121.91%. However, since a matching physical algorithm affects also all physical operators that follow, this penalization factor is propagated to all physical operators that are after the matching physical operator. Therefore, the cost of the default clustering algorithm will also be increased by the same factor of 121.91%. If there were more physical algorithms after the clustering, since they all come after the matching, they would also suffer the same penalization. The cost of the execution plan shown in Figure 4.9 would then be the sum between the default mapping algorithm, the traditional blocking algorithm, and the default clustering algorithm with traditional blocking's penalization factor applied.

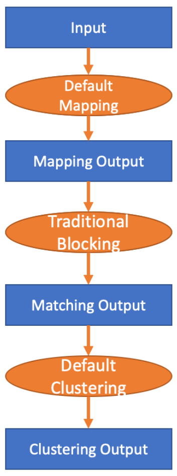


Figure 4.9: Execution plan for a DCP with a mapping, a matching, and a clustering physical operator

4.4 Execution Optimization

Some design choices make working on CLEENEX very hard and some make it difficult to scale. These design choices are: *(i)* the generation of code in runtime, and *(ii)* how the algorithms were implemented.

Generating code in runtime makes debugging very challenging since the code does not exist previously to the data cleaning program execution. Also, it does not allow us to decouple an algorithm from the CLEENEX platform, thus not allowing us to test it individually. Moreover, the introduction of new features is very time-consuming and error-prone.

Another CLEENEX bottleneck is its algorithms. Since this thesis focused more on the Matching operator, this analysis is mainly about its algorithms. Most of the algorithms, that were recently implemented, fail to use programming best practices. This makes those algorithms very strict concerning potential changes. Moreover, some choices regarding the implementation of the algorithms and the structures used to support them make the execution complexity increase without notice, thus also increasing the execution time.

In this section, we address the optimizations that were performed to decrease CLEENEX execution time, more precisely, for the Matching physical algorithms. In Section 4.4.1 we detail the conversion from runtime code to compile code. In Section 4.4.2 we describe the reimplemented matching algorithms.

4.4.1 Conversion from Runtime to Compile Time Code

Moving from runtime to compile-time is not seen as a performance enhancement feature. Nonetheless, changing from runtime to compile-time code achieved a speedup of 50%, i.e., the time needed from the point where the user requests the DCP execution until the time it starts executing, decreased 50% when using compile-time code. With runtime code, in order to start executing a matching physical algorithm CLEENEX took 200ms. In this time it is included the interpretation of the declared data cleaning program, the creation of the runtime code and its compilation. With compile-time code, it takes 100ms, which is the data cleaning program interpretation time.

As said earlier, currently only the matching operator has all of its code available at compile time. The change was not transversal to all operators for two reasons: *(i)* most of the code cannot be reused across operators, and *(ii)* this change is very time-consuming, so we focused on the operator that we were working.

To replace the generation of runtime code in the matching physical operator with compiled code, i.e., code that is present in CLEENEX codebase, we analyzed and divided the generated code into multiple sections. This division enables us to understand possible differences among matching physical operators. We present a pseudo-code of a generic matching physical algorithm generated in runtime on Listing 4.3. There we can identify three sections: *(i)* reading input data and creating output tables (lines 1-3), *(ii)* generating the candidate pairs (lines 4-5), and *(iii)* applying the WHERE clause defined in the logical operator to each record pair (lines 6-12). This pseudo-code represents all the matching physical algorithms.

```
1  // Section 1
2  Record [] input = readInputTables()
3  createOutputTables();
4  // Section 2
5  Pairs recordPairs = generateCandidateRecordPairs(input)
6  foreach pair in recordPairs {
7     //Section 3
8     boolean whereClause = getLogicalOperatorWhereClauseResult()
9     if whereClause is true {
10       insertPairInOutputTable()
11    }
12 }
```

Listing 4.3: Psuedo-code representing the runtime code of a matching physical algorithm

As said previously, all matching physical algorithms share the same pseudo-code since they all perform the same actions. Therefore, it is straightforward to create a compile-time version. All physical algorithms can share the same code differing only on how they generate the candidate record pairs, i.e., in the second section in Listing 4.3.

4.4.2 Matching Algorithms Optimization

CLEENEX matching algorithms were the work of another master thesis [13]. That work made available in CLEENEX five matching operators, Adaptive Sorted Neighborhood Join, Canopy Clustering, Sorted Neighborhood Join, Inverted Index Sorted Neighborhood Join, and Traditional Blocking. One of the goals of this thesis was to parallelize and distribute them. However, when studying CLEENEX it was clear that the matching algorithms were being a major bottleneck in the execution. For example, with a small input of 500 records, algorithms such as the Traditional Blocking and Adaptive SNJ took 26 minutes, and 57 minutes, respectively. This is not a good execution time, for two reasons: *(i)* the Cartesian

Product is achieving better times, and *(ii)* if we remember that our goal was to support datasets with millions of records, taking so much time with 500 input records is undesirable. Moreover, some of them were outputting an unexpected number of records, so there were also some concerns regarding the generation of candidate record pairs.

After some investigation, we concluded that the majority of the performance issues are a consequence of the data structures that are used to support the execution of the algorithms. In what concerns the generated candidate pairs, we have two problems: *(i)* the organization of the records at the time the dataset is sorted affects more than expected the candidate pairs generated for the Sorted Neighborhood Join algorithms, and *(ii)* for non-self-matching there was a bug in the definition of the algorithms.

Changes in the Algorithms

The major refactoring on the Sorted Neighborhood Join, henceforth SNJ, and Adaptive SNJ, henceforth ASNJ, is on the way that the window is created and the data structures used to support their execution.

We can distinguish two different algorithms that enable the generation of candidate record pairs: *(i)* the one that iterates through the dataset, and *(ii)* the one that creates a window and generates the candidate pairs.

For the second algorithm, there was a need to refactor it completely. The actual Java code that was available in CLEENEX is shown in Listing 4.4, being similar for both SNJ and ASNJ. The algorithm is supported by an ArrayList structure that gathers all candidate pairs generated throughout the algorithm's execution. The first good practice error is the usage of the ArrayList instead of its supertype, List. This decision denies a straightforward usage of important Java features such as the Stream API [3]. The Stream API can be used to perform lazy[4] data operations in parallel. Among other possible data structures, the Stream API allows to return a List, but not an ArrayList. Although there is a workaround so that the ArrayList can be used, it would imply consuming double the memory unnecessarily, which is undesirable.

```java
public ArrayList<Map<String, Record>> getCandidatePairs(ArrayList<Map<String, Record>> pairs) {
  for(int i = index; i < index + length; i++) {
  Record t1Record = table.get(i).getRecord();
    for(int j = i + 1; j < index + length; j++) {
      Record t2Record = table.get(j).getRecord();
      if(!t1Record.getTableName().equals(t2Record.getTableName())) {
        Map<String, Record> pair = new HashMap<>();
        pair.put(t1Record.getTableName(), t1Record);
        pair.put(t2Record.getTableName(), t2Record);
        if(!pairs.contains(pair)) {
          pairs.add(pair);
        }
      }
```

[3]https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
[4]https://en.wikipedia.org/wiki/Lazy_evaluation

```
14      }
15    }
16    return pairs;
17  }
```

Listing 4.4: Java code to generate candidate pairs in Sorted Neighborhood Join and Adaptive SNJ

The second, and most impactful error, is the choice of a List to gather the already generated candidate pairs. As can be seen in Table 4.4, the List data structure has an $O(N)$ CPU cost when searching for data. This compares with Map's $O(1)$ CPU cost. As we can observe at line 10 of Listing 4.4, for each pair of records a search in the list structure is performed, having a huge impact on the algorithm's performance. By replacing the List with a Map data structure, performing a search is unnoticeable performance-wise.

Data Structure	Read	Search	Write
List	$O(N)$	$O(N)$	$O(1)$
Map	$O(1)$	$O(1)$	$O(1)$

Table 4.4: Algorithmic Complexity of List and Map Data Structures

To use the Map data structure we need to define a key to represent the pair of records. This key needs to be generated in a way that guarantees that no pair is compared twice independently of having a different order, i.e., if there are two records A and B, the pair $A - B$ has the same key as the pair $B - A$. The key couldn't be generated from the blocking key since that is not unique among the records. Thus, the key is created from the records' primary key. In the current implementation, the primary key is given by the user through a hint. The hint has the syntax $pk = "key_A, key_B"$, where key_A is the key for the first declared table, and key_B, the key for the second one. Alternatively, the user can also use the singular syntax $pk = "key"$. In this case, the algorithm assumes that key is used for both the first and second table declared, being useful as a shortcut when declaring a self-matching. The key that represents a pair of records in the map structure is generated as follows. First, CLEENEX queries the database to retrieve the values of the primary key columns declared in the pk hint. Then, those values are sorted alphabetically. For example, assume that there are two records, A and B. If record A primary key value is "235" and record B primary key value is "666", then the generated key would be "235_666".

Cross Matching Operator Optimizations

The refactored matching physical algorithms, i.e., the Sorted Neighborhood Join, the Adaptive SNJ, and the Traditional Blocking read the input datasets differently from the remaining physical algorithms. In case of a self matching, i.e., a matching whose input tables are the same, that input dataset is only read once. In the previous implementation, even if on a self matching, CLEENEX would request to read twice the same input table. In the refactored implementation, a physical algorithm can detect that it is executing a self matching by analyzing the input tables' names instead of the alias names. An example of a self matching declaration is shown in Listing 4.5. The tables alias' are defined in the $FROM$ clause (line 2). In that clause, the user declares that $T1$ will serve as an alias for table $PubAuthorNames$, and that $T2$ will be an alias for that same table.

```
CREATE MATCHING SimilarAuthors
FROM PubAuthorNames T1, PubAuthorNames T2
% scale-up = "SNJ" scale-up key = "lastname" window = 3 %
LET sim =similarAuthors(T1.firstname, T1.lastname, T2.firstname, T2.lastname)
WHERE sim > 0.95 AND T1.authorid <> T2.authorid
{
SELECT  T1.authorid as authorid1,
T1.firstname as firstname1,
T1.lastname as lastname1,
T2.authorid as authorid2,
T2.firstname as firstname2,
T2.lastname as lastname2
}
```

Listing 4.5: Example of a self-matching declaration

Chapter 5

Experimental Validation

In this chapter, we describe the experiments that enable to validate the optimizer proposed and described in Chapter 4. The goal is to evaluate the capability of the optimizer to choose the best execution plan for a given data cleaning program. This experimental validation allows us to see if the optimizer is creating the expected plans and selecting the cheapest one. Furthermore, we evaluate the performance gain achieved with the introduction of the optimizer.

In Section 5.1, we describe the setup used throughout the experimental validation, namely, the datasets, the data cleaning programs, and the metrics used to evaluate the results obtained. Then, in Section 5.2 we evaluate the results of the cost model, where we compare the estimated values with the real ones. Finally, in Section 5.3, we analyze the impact of the optimizations made in what regards the runtime code to compile time code in the matching operator and the matching algorithms optimizations.

5.1 Experimental Setup

This section describes the datasets used, the data cleaning programs that were executed as well as the metrics we use to perform the evaluation.

The experiments were performed in a Macbook Pro 2017 having 4 cores with 2.8GHz and 16 GB of main memory (RAM) of 2,133 MHz LPDDR3. The operating system is macOS Catalina version 10.15.6.

5.1.1 Datasets

For the evaluation of the optimizer's performance, we used multiple variations of the same dataset, the CIDS Publication. This dataset is based on the gold standard dataset CORA [1]. However, the CIDS Publication only contains a subset of CORA dataset, i.e., it contains fewer records, thus it does not have the gold standard.

The CIDS publication dataset originally contains 481 records. To perform the experiments that allow us to evaluate our solution we needed datasets with more input records, in our case, up to 250,000

[1] https://hpi.de/naumann/projects/repeatability/datasets/cora-dataset.html

entries. There is the need to maintain coherency between the contents of those datasets since we need to guarantee as much as possible that the results are only influenced by the input size growth and not by the content. Therefore, we used the CIDS Publication dataset and created several variations of it. These variations have 500, 1,000, 5,000, 25,000, 100,000, and 250,000 input records. To generate such records from the original 481, we created a data generator. This data generator saves the content of each column from all dataset records in memory. Then, to create a new record, it selects randomly from each in-memory column a value and mounts the record. For example, consider a dataset with two records, X and Y, and two columns, A and B. An example of a record Z generated by the data generator is $record\ Z : X_A, Y_B$, i.e., the new record Z, assumes the value of column A in record X, and the value of column B in record Y.

The CIDS Publication dataset is described by the following attributes:

- **pid**: identifies uniquely an entry in this dataset;
- **aid**: identifies uniquely a group of authors;
- **title**: title of the publication;
- **authors**: authors of the publication;
- **year**: year of the publication;
- **bibtex**: represents the bibtex code of the publication;
- **linkgoogle**: link for the publication in Google Scholar;
- **cits**: number of citations for the publication;
- **citslink**: link to access the citations of the publication;
- **citsns**: number of citations that cite all publication authors at the same time;
- **citsnslink**: link for citations that cite all publication authors at the same time;
- **citsslink**: link that gathers all citations that every publication author has;

5.1.2 Data Cleaning Program

Once again, to maintain consistency between different results, there is the need to use a single Data Cleaning Program (DCP) across all experiments.

The DCP used in our evaluation is the one shown in Figure 5.1. It receives as input one of the datasets that were generated from the CIDS Publication dataset detailed in Section 5.1.1, as an example, the DCP in Figure 5.1 receives the dataset variation with 1,000 input records, *cidspub1k*. The oval-shaped objects represent logical operators. The rectangular-shaped objects the input table, and the logical operators' output. In the remainder of this chapter, we refer to the logical operators by the name of their output table, e.g., the logical operator *Mapping1* is referred to as *AuthorsByPublication*.

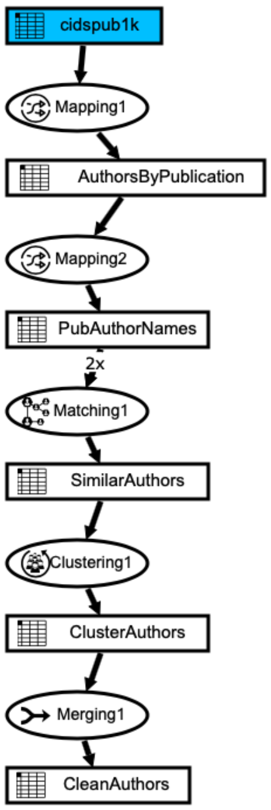


Figure 5.1: Data Cleaning Graph that represents the DCP used in the experiments

The mapping *AuthorsByPublication* logical operator receives an input record, and for each author in the list of authors of the publication, it creates a new record and associates it to the same publication. When dividing the authors, it associates a unique identifier to each of them, the uid. This unique identifier is retrieved by a User Defined Function (UDF). Also, it separates the author name in first and last name. For example, the publication A, identified by the $pid = 1$ and $aid = 999$, whose authors column value is $Rodrigo, AdelinoB, MariaRodrigues$ generates three records, one for each author, $\{A, pid = 1, aid = 999, uid = 1, authorFirstName = Rodrigo, authorLastName = B\}$, $\{A, pid = 1, aid = 999, uid = 2, authorFirstName = Adelino, authorLastName = B\}$, and $\{A, pid = 1, aid = 999, uid = 3, authorFirstName = Maria, authorLastName = Rodrigues\}$.

The second mapping, *PubAuthorNames*, standardizes the author names so that they do not contain white spaces or special characters. The output size is the same as the input size.

The matching logical operator, *SimilarAuthors*, defines as blocking key, i.e., the key that represents a record in the matching, the column $authorFirstName$ generated in $AuthorsByPublication$. In case a Sorted Neighborhood Join or Inverted Index SNJ physical algorithm is used, the window size is of 3 records. The matching will be a self-matching, where the input table is the output of mapping that precedes it, i.e., $PubAuthorNames$. The output size depends on which physical algorithm is executed since each one has a different expected output size.

The clustering logical operator, *ClusterAuthors*, gathers the pairs of records produced by the preceding matching operator and applies the Transitive Closure, which allows to discover similar data that is indirectly related. For example, given three records A, B, and C, if A and B are similar and B and C are also similar, then A and C are also similar, even though they are not directly related, i.e., there is no pair with those two records. The output is dependent on the matching output.

The merging operator *CleanAuthors* is the last logical operator in the DCP. Its goal is to select one record for each cluster it receives as input. The selection of that record is delegated to a UDF defined in the merging logical operator. In this DCP, the record that is selected from each cluster is the one whose combination of $authorFirstName$ and $authorLastName$ has the biggest length, i.e., the highest count of characters. For example, if a cluster has two records, $\{A, authorFirstName = Ana, authorLastName = R$, and $\{B, authorFirstName = Michael, authorLastName = Jackson$, record B will be selected since the length of its columns value is 14 characters which compares with record's A length of 4.

5.1.3 Metrics

In this section, we describe the metrics used to evaluate the cost model and the optimizations performed in CLEENEX. More specifically, these metrics aim at evaluating the difference between the real output size and the estimated output size. Moreover, the metrics also allow us to evaluate the performance gain achieved with the optimizations.

Error Rate

We need to evaluate the error rate between the estimated output size and the real output size of a physical operator. For all physical operators except the matching, the error rate is given by the formula in Equation 5.1. An error rate less than 1.0 means that the estimated value is higher than the operator real output, greater than 1.0 the estimated value is smaller than the operator real output, and when the error rate is 1.0 the estimated value and operator real output are the same.

$$Error\ Rate = \frac{Physical\ Operator\ Real\ Output}{Estimated\ Output} \tag{5.1}$$

For the matching physical operator and its physical algorithms, the error rate is computed by using the formula in Equation 5.2. We can have the same interpretation as in the previous equation, but instead of comparing against the physical operator real output, we compare against the number of candidate pairs generated. What differentiates the matching physical algorithms is how they generate the candidate record pairs and how many they generate. This is why we evaluate the error rate by analyzing the candidate record pairs.

$$Error\ Rate_{Matching} = \frac{Candidate\ Pairs\ Generated}{Estimated\ Pairs\ Generated} \tag{5.2}$$

Performance Speedup

To measure the performance gain achieved with the optimizations made, we use the performance speedup formula in Equation 5.3. The T_{old} parameter is the execution time achieved before the optimizations were performed, and T_{new} the execution time after introducing the optimizations. A speedup less than 1.0 means that the old implementation execution time was lower than the new one, speedup greater than 1.0 the new implementation execution time is lower than the old one, and a speedup of 1.0 means that both execution times are identical.

$$Speedup = \frac{T_{old}}{T_{new}} \tag{5.3}$$

5.2 Cost Model

The Cost Model affects the overall execution of a data cleaning program since it is essential to choose which execution plan should be used. In the cost model, for each operator, we estimate the algorithmic cost, that is, the CPU cost, the I/O cost, and the estimated output size.

The DCP in Figure 5.1 generated six execution plans. In these plans, all physical operators use their default physical algorithm but the matching physical operator. The execution plans are enumerated in Table 5.1, and are used in the remainder of this section to evaluate how the cost model estimations compare with the real values.

Plan Number	**SimilarAuthors** Physical Algorithm
1	Cartesian Product
2	Sorted Neighborhood Join
3	Adaptive SNJ
4	Canopy Clustering
5	Inverted Index SNJ
6	Traditional Blocking

Table 5.1: Generated Execution Plans

In Section 5.2.1 we analyze the cost of the execution plans listed in Table 5.1, detail the differences between the plans and explain how the matching physical operator affects the execution plan. Then, in Section 5.2.2 we compare the real output size against the estimated output size for each algorithm and verify what is the error rate. Finally, we discuss the results achieved in Section 5.2.3.

5.2.1 Plans Cost

Tables 5.2 to 5.7 show the estimated cost for each plan listed in Table 5.1. In those tables, for each physical algorithm in the plan, we list its *CPU Cost*, *I/O Cost*, the sum of these two costs without the Penalization Factor (PF), *Total Cost (w/o PF)*, and with PF, *Total Cost (w/ PF)*, and the *Estimated Output* according to the formulas presented in Section 4.3.1. Finally, at the bottom of the tables, we present the *Plan Cost*. This plan cost is the sum of all physical algorithm costs with the penalization factor, i.e., the sum of column *Total Cost (w/ PF)*. Recall that the penalization factor, described in Section 4.3.3, penalizes the matching operator's final cost and every operator that is after it in the execution plan.

Table 5.2 refers to the plan number one in Table 5.1. This plan uses the default matching algorithm, Cartesian Product. Since this algorithm is lossless, there is no penalization factor. This means that the plan cost of 1.93×10^7 is the sum of all physical algorithm costs.

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Cartesian Product (*SimilarAuthors*)	4,000,000	66,306	4,066,306	4,066,306	1,000,000
Default Clustering (*ClusterAuthors*)	1,000,000	8,119,280	9,119,280	9,119,280	1,000,000
Default Merging (*CleanAuthors*)	6,000,000	154,473	6,154,473	6,154,473	300,000
Plan Cost	1.93×10^7				

Table 5.2: Cost model by physical algorithm for plan 1 in Table 5.1

The cost of the second plan in Table 5.1 is shown in Table 5.3. The matching physical algorithm used in this plan is the Sorted Neighborhood Join. This algorithm has a penalization factor of 124,96%, meaning that the sum of its CPU and I/O cost, the *Total Cost (w/o PF)* column value in table 5.1, 7,116, will be multiplied by that factor, resulting in 8,892,15, as shown in column *Total Cost (w/ PF)*. Moreover, as said earlier, this penalization factor is propagated to the following physical algorithm's cost. The plan cost is 143,072.76.

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Sorted Neighborhood Join (*SimilarAuthors*)	6,602	514	7,116	8,892.15	7,997
Default Clustering (*ClusterAuthors*)	7,997	64,924	72,921	91,122.08	7,997
Default Merging (*CleanAuthors*) CleanAuthors	31,211	1,235	32,446	40,544.52	2,399
Plan Cost	143,072.76				

Table 5.3: Cost model by physical algorithm for plan 2 in Table 5.1

The cost of the plan number three in Table 5.1 is shown in Table 5.4. In this plan, the Adaptive SNJ is used to execute the matching physical operator. This algorithm has a penalization factor of 121.91%. The plan cost after applying the penalization factor to the matching, clustering, and merging physical algorithms data operators is 4,218,704.30.

The Canopy Clustering is the matching algorithm used in Table 5.1 plan number three. The cost

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Adaptive SNJ (*SimilarAuthors*)	6,602	514	7,116	8,675.12	235,650
Default Clustering (*ClusterAuthors*)	235,650	1,913,304	2,148,954	2,619,789.82	235,650
Default Merging (*CleanAuthors*)	1,265,974	36,401	1,302,375	1,587,725.36	70,695
Plan Cost	4,218,704.30				

Table 5.4: Cost model by physical algorithm for plan 3 in Table 5.1

model estimations for this plan are shown in Table 5.5. Curiously, this plan cost is 267% higher than the one with the default matching algorithm, Cartesian Product, shown in Table 5.2. Two factors may contribute to such a high cost: *(i)* the Canopy Clustering CPU cost, $N^2 \times log(N)$, grows quicker than that of Cartesian Product N^2, and *(ii)* contrary to the Cartesian Product, the Canopy Clustering has a penalization factor of 193.66% that almost doubles its real cost, thus affecting the plan cost. This high penalization factor is mainly because it is a bad performer.

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Canopy Clustering (*SimilarAuthors*)	13,204,119	514	1.32×10^7	25,572,092.27	878,906
Default Clustering (*ClusterAuthors*)	878,906	7,136,080	8,014,986	15,521,821.89	878,906
Default Merging (*CleanAuthors*)	5,224,166	135,766	5,359,932	10,380,044.31	263,671
Plan Cost	5.15×10^7				

Table 5.5: Cost model by physical algorithm for plan 4 in Table 5.1

Table 5.6 shows the cost for the plan number five in Table 5.1. This plan uses the Inverted Index SNJ as the matching physical algorithm. Due to being underperforming when compared with the remaining matching physical algorithm, the Inverted Index SNJ has a huge penalization factor of 192.81%. However, when compared with the Canopy Clustering, its CPU cost, $N \times log(N)$ grows slower, its penalization factor is less 0.85% than the Canopy's, and finally, the estimated output size is much smaller. Note that the estimated output size is used in the CPU and I/O cost computations in the following physical algorithm, the default clustering algorithm. A smaller estimated output size means that the cost of reading and iterating that output will be smaller.

The last plan in Table 5.1 has its costs shown in Table 5.7. Traditional Blocking is used to execute the matching physical operator. This physical algorithm has a penalization factor of 121.91%. According

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Inverted Index SNJ (*SimilarAuthors*)	6,602	514	7,116	13,720.36	32,934
Default Clustering (*ClusterAuthors*)	32,934	267,400	300,334	579,073.99	32,934
Default Merging (*CleanAuthors*)	148,784	5,087	153,871	296,678.68	9,880
Plan Cost	891,987.02				

Table 5.6: Cost model by physical algorithm for plan 5 in Table 5.1

to the cost model, this physical algorithm is estimated to generate a considerable amount of candidate record pairs, as the value of the *Estimated Output* column of Table 5.7 shows. Moreover, this algorithm is considered a good performer. These two facts help explain why the penalization factor is low (121.91%). Nonetheless, this plan also has a higher cost than the one with the default matching physical algorithm, Cartesian Product. The reasoning behind this high cost is the same as for the Canopy Clustering since both physical algorithms share the same CPU cost formula.

Physical Algorithm	**CPU Cost**	**I/O Cost**	**Total Cost** (*w/o PF*)	**Total Cost** (*w/ PF*)	**Estimated Output**
Default Mapping (*AuthorsByPublication*)	1,000	257	1,257	1,257	1,000
Default Mapping (*PubAuthorNames*)	1,000	257	1,257	1,257	1,000
Traditional Blocking (*SimilarAuthors*)	13,204,119	514	1.32×10^7	16,092,120	235,650
Default Clustering (*ClusterAuthors*)	235,650	1,913,304	2,148,954	2,619,789.82	235,650
Default Merging (*CleanAuthors*)	1,265,974	36,401	1,302,375	1,587,725.36	70,695
Plan Cost	2.03×10^7				

Table 5.7: Cost model by physical algorithm for plan 6 in Table 5.1

The list of plans ordered by ascending cost is available in Table 5.8. As can be seen in that table, the cheapest plan uses the Sorted Neighborhood Join matching physical algorithm. Although this physical algorithm is the least efficient, it is also the best performer. Recall that the efficiency was measured in terms of the number of generated candidate record pairs when compared with the default matching physical algorithm, the Cartesian Product. Since the SNJ is the least efficient matching algorithm, it generates fewer record pairs than any other matching algorithm. This means that physical algorithms executed in the context of physical operators following the matching operator will have a lower I/O and CPU cost. Therefore, they need to read and iterate through fewer records, thus having a lower impact on the plan cost. The main surprise of this list is the positioning of the Inverted Index SNJ. Despite its high penalization factor, due to its low CPU cost and misleading estimated output size, it was able to achieve second place.

Order	Plan Number	Matching Algorithm
1	2	Sorted Neighborhood Join
2	5	Inverted Index SNJ
3	3	Adaptive SNJ
4	1	Cartesian Product
5	6	Traditional Blocking
6	4	Canopy Clustering

Table 5.8: Plans from Table 5.1 sorted by ascending plan cost

5.2.2 Output Size

The output size estimation is an essential part of the optimizer. Recall that the estimated output size of a physical operator is the estimated input size of the physical operator that follows it. Therefore, the output size estimation affects the following physical operator, not the physical operator where the estimation is performed. Within this section, we want to understand what is the error rate of the cost model estimations.

To evaluate the output size estimation, we used the same DCP represented in Figure 5.1, but varied its input size between 500, 1,000, 5,000, 25,000, 100,000, and 250,000. In this experiment, for the matching physical operator, we evaluated all of its physical algorithms results. Recall that we use the name of the logical operator output to represent the logical and physical operator (e.g., *Mapping1* is referred to as *AuthorsByPublication*). In what regards the physical operators that follow the matching, *ClusterAuthors*, and *CleanAuthors*, we evaluated the results for the cheapest execution plan only, i.e., the one that uses Sorted Neighborhood Join as matching physical algorithm, as detailed in Section 5.2.1.

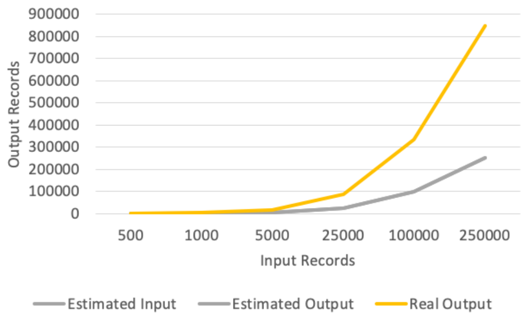
In the remaining of this section, for each logical operator in Figure 5.1 DCP, we evaluate the output size estimation error rate. We perform this evaluation for each physical algorithm capable of executing those logical operators.

Mapping Operators

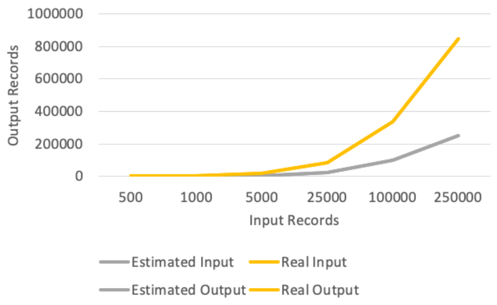
We start by analyzing the first two mapping logical operators, executed by the default matching physical algorithm. Figure 5.2 illustrates how the estimated output size compares with the real output size. Moreover, to evaluate how the input size estimation affects the output size estimation, we also present data regarding the estimated and real input size for both physical algorithms.

Figure 5.2a graphic shows that despite having no differences between real and estimated input, the real output size is wrongly estimated for the first mapping, *AuthorsByPublication*. According to mapping's cost model, it is expected one output record for each input one, which was not verified, i.e., for each input record, more than one record was output. The physical algorithm had an error rate of 3.37, i.e., for each input record, on average, 3.37 records were output. As described in Section 4.3.1, the mapping semantics allows this operator to have an unpredictable output size, thus being difficult for a fixed cost model, i.e., a cost model that does not change dynamically depending on the DCP declaration, to estimate that output correctly.

In the second mapping, *PubAuthorNames*, we can see how the output size estimation of *AuthorsByPublication* affects the following physical algorithm estimations. As represented in Figure 5.2b, the estimated input, i.e., *AuthorsByPublication* estimated output, and output differ from the real input and output. This difference is solely explained by the estimation error in the previous mapping operator. This explains why *PubAuthorNames* has the same error rate as *AuthorsByPublication*, i.e., 3.37.



(a) Cost model results for default mapping physical algorithm (logical operator *AuthorsByPublication*)



(b) Cost model results for default mapping physical algorithm (logical operator *PubAuthorNames*)

Figure 5.2: Cost model results for the first two mapping physical operators

Matching Operator

After the mapping operations, we have the matching logical operator. The corresponding matching physical operator can be executed by six different physical algorithms. We analyzed the cost model for each one. Note that the estimated output refers to the estimated number of candidate pairs, as stated earlier in Section 4.3.1.

For the Sorted Neighborhood Join, Figure 5.3 shows that the cost model is consistently accurate across the various input sizes tested. As shown in Table 5.9, the estimated pairs generated is not far from the effective number of candidate pairs generated, i.e., the number of candidate record pairs output by the physical algorithm. The cost model achieves an error rate of 0.83, i.e., for every 100 estimated candidate record pairs, the physical algorithm, in reality, outputs 83.

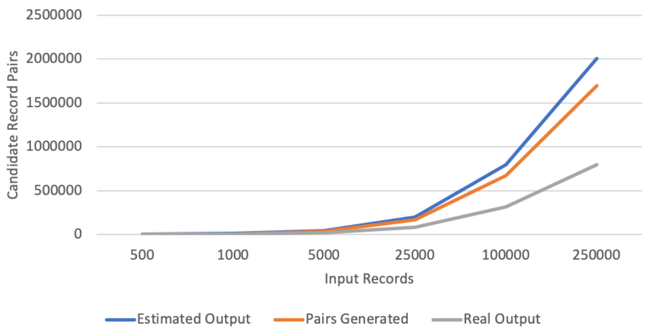


Figure 5.3: Cost model results for *SimilarAuthors* (Matching) with Sorted Neighborhood Join

	500	1,000	5,000	25,000	100,000	250,000
Estimated Pairs Generated	3,997	7,997	39,997	199,997	792,005	2,001,869
Candidate Pairs Generated	3,355	6,775	33,749	170,445	670,639	1,691,459
Matching Real Output	1,493	3,086	15,720	79,826	313,550	791,020
Error Rate	0.84	0.85	0.84	0.85	0.85	0.84

Table 5.9: Cost model results summary for *SimilarAuthors* (Matching) with Sorted Neighborhood Join

The Inverted Index SNJ physical algorithm has a similar CPU cost to SNJ. However, it can detect more pairs than that algorithm. In fact, only the Cartesian Product generates more candidate record pairs. Although producing more candidate record pairs does not mean that more duplicates are detected, it does increase the chances of finding more duplicates, since more records are compared. In what concerns the cost model, as Figure 5.4 shows, the number of estimated candidate record pairs is approximately 2,469% smaller than the actual generated candidate record pairs. This corresponds to an error rate of 24.69, as seen in Table 5.10. Therefore, the formula for the output size estimation, extracted from [6], is not able to precisely estimate for the Inverted Index SNJ. Alternatively, the authors of [6] propose a formula using the Zipf distribution, instead of the one that our cost model is using, normal distribution. However, for the Inverted Index SNJ, to compute the estimations using the Zipf distribution, we would have an approximate CPU cost of $O(2N^2 + N^3)$. Such a high CPU cost, even for small input datasets, is undesirable. To avoid such a high computation cost, we preferred to maintain the normal distribution high error rate.

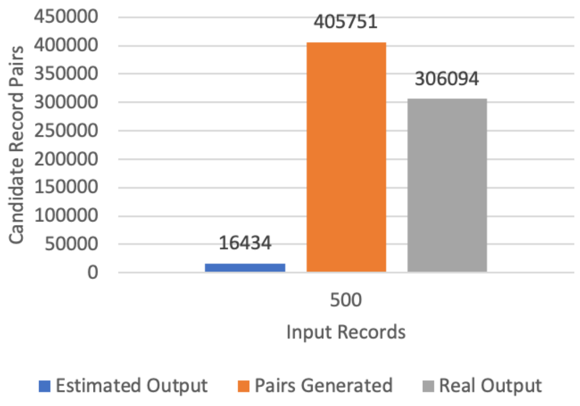


Figure 5.4: Cost model results for *Similar Authors* (Matching) with Inverted Index SNJ

	500
Estimated Pairs Generated	16,434
Candidate Pairs Generated	405,751
Matching Real Output	306,094
Error Rate	24.69

Table 5.10: Cost model results summary for *SimilarAuthors* (Matching) with Inverted Index SNJ

The cost model results for the Adaptive SNJ are illustrated in Figure 5.5. In that figure, it is possible to draw two conclusions: *(i)* the estimated output is undesirably far from the candidate pairs, and *(ii)* the candidate pairs are surprisingly near the real output, meaning that there is no much filtration happening, which contrasts with the SNJ behavior. As Table 5.11 shows, the real output is, on average, 91% of the generated pairs, whereas the generated pairs are 284% more than the estimation.

The cost model results for the Adaptive SNJ, represented in Figure 5.5, show that the estimated candidate record pairs are still far from the real number of generated candidate record pairs. Curiously, the Adaptive SNJ candidate record pairs are very near to the matching physical algorithm output, i.e., after applying the filtering phase defined in the logical operator $WHERE$ clause. A possible justification for this phenomenon is that to create the dynamic window, the Adaptive SNJ uses a similarity function to filter every pair of records. The dynamic window keeps increasing while the similarity value between two adjacent records is above a given threshold. The filtering phase defined in the logical operator through the $WHERE$ clause in the DCP of Figure 5.1, performs similar filtering to that of the Adaptive SNJ, i.e., it uses a similarity function to test every pair of records. According to Table 5.11, the real output is, on average, 91% of the generated candidate record pairs, whereas the generated pairs are 284% more than the estimated ones, i.e., the average error rate is 2.84.

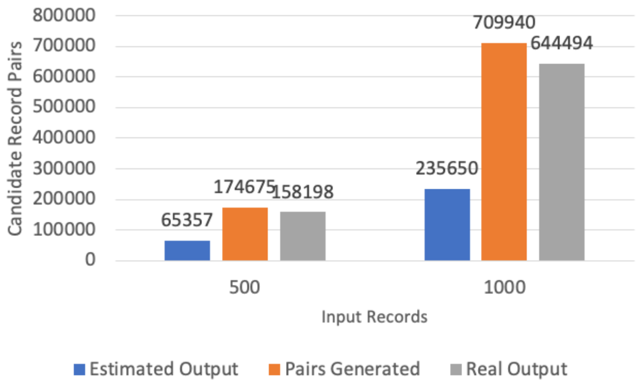


Figure 5.5: Cost model results for *SimilarAuthors* (Matching) with Adaptive SNJ

	500	**1,000**
Estimated Pairs Generated	65,357	235,650
Candidate Pairs Generated	174,675	709,940
Matching Real Output	158,198	644,494
Error Rate	2.67	3.01

Table 5.11: Cost model results summary for *SimilarAuthors* (Matching) with Adaptive SNJ

As stated earlier in Section 4.3.1, both Adaptive SNJ and Traditional Blocking share their output estimation formulas. Moreover, these formulas assume that the input dataset uses a *Zipf Distribution* [30]. The Zipf distribution assumes that in a list of words ordered by their frequencies, the word at position p has a relative frequency of $1/p$. The results presented in Figure 5.5 and Table 5.11 use the Zipf distribution. With the normal distribution, the number of generated candidate record pairs for an input dataset of 500 records would be 833 pairs, which compares with the estimated 65,357 with the Zipf distribution. The results with the normal distribution for the Adaptive SNJ are shown in Figure 5.6. On average, the error rate with the normal distribution is 317.91, i.e., almost 112 times bigger than with the Zipf distribution.

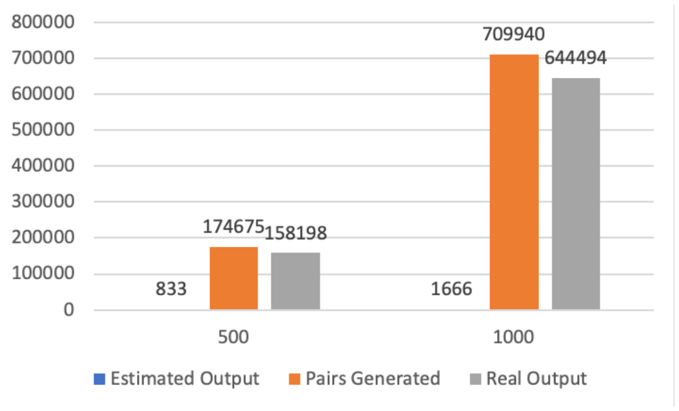


Figure 5.6: Cost model results for *Similar Authors* (Matching) with Adaptive SNJ using Normal Distribution

The results for the Traditional Blocking's cost model are identical to the Adaptive SNJ's, reported in Table 5.11. The only difference is that for 1,000 input records, Traditional Blocking produced less 2,782 pairs than the Adaptive SNJ, as can be seen in Figure 5.7. However, both estimations and real matching output, i.e., after the filtering phase, are identical, thus supporting the choice of sharing the cost model between them. Moreover, both have the same error rate, as reported in Table 5.12.

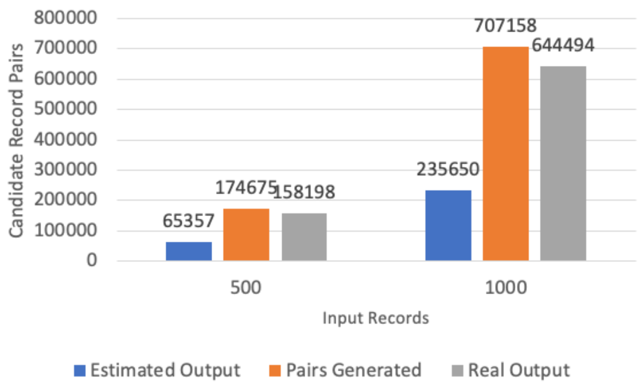


Figure 5.7: Cost model results for *SimilarAuthors* (Matching) with Traditional Blocking

	500	**1,000**
Estimated Pairs Generated	65,357	235,650
Candidate Pairs Generated	174,675	707,158
Matching Real Output	158,198	644,494
Error Rate	2.67	3.01

Table 5.12: Cost model results summary for *SimilarAuthors* (Matching) with Traditional Blocking

The Cartesian Product is the operator with the highest output size estimation. This algorithm compares all records against each other. It does not perform any optimization, i.e., does not filter pairs, as the remaining matching physical algorithms. For example, the other matching algorithms, for records A and B, only create the pair that appears first, either $A - B$ or $B - A$, whereas the Cartesian Product creates both. Although the Cartesian Product behavior is the most predictable from all the matching physical algorithms, as shown in Figure 5.8, the estimation of generated candidate record pairs is not

100% accurate. As Table 5.13 shows, the average error rate is 11.38. The reason for this lack of accuracy is due to the estimation error made in the previous mapping physical algorithms. Recall that the mapping physical operator that precedes the matching, *PubAuthorNames*, for an input dataset of 1,000 records, has an estimated output of 1,000 records. However, the real output size, and therefore, the real input size of the matching physical operator and its algorithms, is 3,389 records, as illustrated in Figure 5.2b. If there were no errors in the previous estimations, the Cartesian Product, for an input dataset of 1,000 input records would estimate 11,485,321 candidate record pairs, having an error rate of 1.0, i.e., the estimations and real values are identical, meaning that the cost model is accurate.

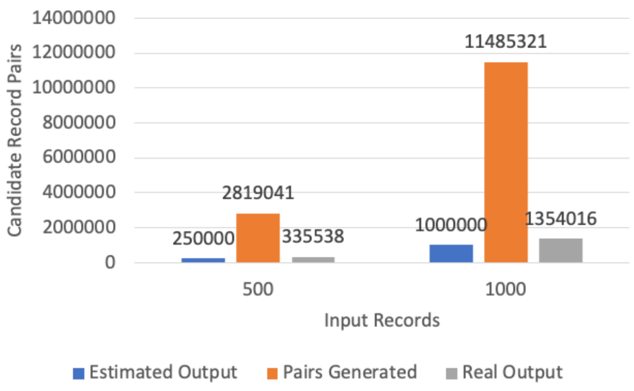


Figure 5.8: Cost model results for *SimilarAuthors* (Matching) with Cartesian Product

	500	**1,000**
Estimated Pairs Generated	250,000	1,000,000
Candidate Pairs Generated	2,819,041	11,485,321
Matching Real Output	335,548	1,354,016
Error Rate	11.27	11.48

Table 5.13: Cost model results summary for *SimilarAuthors* (Matching) with Cartesian Product

The results achieved for the Canopy Clustering cost model are satisfactory, since as shown in Figure 5.9, the estimations made are only 1.63 times smaller than the real values, i.e., the error rate, as reported in Table 5.14, is just 1.63.

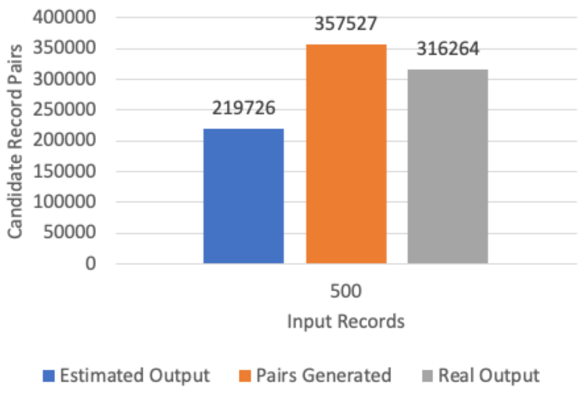


Figure 5.9: Cost model results for *SimilarAuthors* (Matching) with Canopy Clustering

	500
Estimated Pairs Generated	219,726
Candidate Pairs Generated	357,527
Matching Real Output	316,264
Error Rate	1.63

Table 5.14: Cost model results summary for *SimilarAuthors* (Matching) with Canopy Clustering

Clustering Operator

The clustering physical operator only has one physical algorithm, the default. The evaluation performed for this default clustering physical algorithm assumes that the matching physical algorithm performed previously was the Sorted Neighborhood Join. To evaluate the results for the clustering algorithm, we considered the datasets whose input size was 500, 1,000, and 5,000 records.

According to the clustering semantics detailed in 4.3.1, it is expected one output record for each input record, as reflected in the overlapping estimated input and output curves in Figure 5.10. This expectation proves to be accurate when we analyze the real input and output curves since they also overlap, meaning that the expected behavior is followed. However, as reported in Table 5.15, the average error rate is not 1.0 but 0.41. As in the Cartesian Product estimations, the results are affected by estimation errors made in previous physical algorithms.

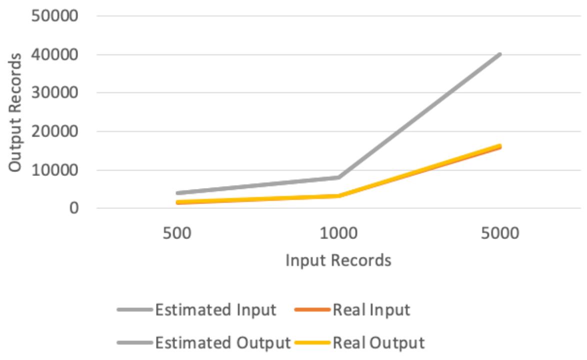


Figure 5.10: Cost model results for clustering default physical algorithm (logical operator *ClusterAuthors*)

	500	1,000	5,000
Estimated Input	3,997	7,997	39,997
Real Input	1,493	3,086	15,720
Estimated Output	3,997	7,997	39,997
Real Output	1,613	3,281	16,390
Error Rate	0.40	0.41	0.41

Table 5.15: Cost model results summary for clustering default physical algorithm (logical operator *ClusterAuthors*)

Merging Operator

As for the default clustering physical algorithm evaluation, for the default merging algorithm, we also used the same three datasets with 500, 1,000, and 5,000 input records.

	500	1,000	5,000
Estimated Input	7,994	15,994	79,994
Real Input	1,613	3,281	16,390
Estimated Output	2,398	4,798	23,998
Real Output	120	195	670
Error Rate	0.05	0.04	0.02

Table 5.16: Cost model results summary for merging operation *CleanAuthors*

In the DCP used for the experiments made in this chapter, represented in Figure 5.1, the merging logical operator defines a User Defined Function (UDF) to select which record represents a records' cluster, as described in Section 5.1.2. UDFs are a black box to CLEENEX, thus being unable to estimate both its cost or output accurately. Therefore, the cost model created estimates the output size as a fixed factor over the estimated input size. This factor is 30% of the physical algorithm's estimated input, as detailed in Section 4.3.1. As can be seen in Figure 5.11, the estimated and real output are very far, as corroborated by the average error rate of 0.04 reported in Table 5.16. In summary, the merging physical operator, independently of the physical algorithm that executes it, suffers from the same problem as the mapping physical operator: their output is only dependent on the UDF behavior, which is a black box for CLEENEX and its optimizer.

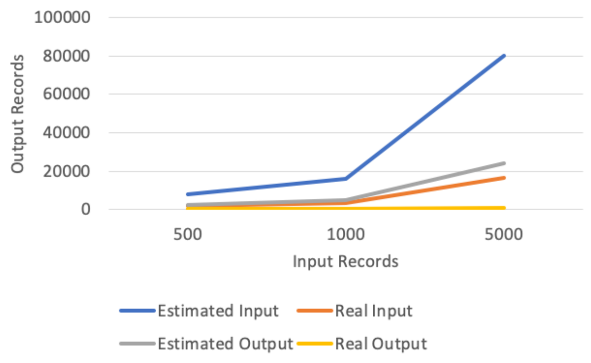


Figure 5.11: Cost model results for default merging physical algorithm (logical operator *CleanAuthors*)

5.2.3 Discussion

As can be verified throughout this section, the cost model estimations are not always on par with the real values, i.e., the average error rate is not 1.0. The average error is 2.37. This means that for a plan that should cost 1,000, the cost model will, on average, evaluate its cost as if it was 2,370.

The main challenge when trying to decrease the average error rate is the dependency of the physical algorithms in User Defined Functions (UDFs). These UDFs are a black box to CLEENEX, and consequently, its optimizer. Most results were affected by the bad output estimation of the first two mapping physical operators, thus explaining why the error rate for some physical algorithms is so far from 1.0.

5.3 Matching Algorithms Optimizations

This thesis provided several optimizations, even though it was not one of its goals. Some of them are not quantifiable, such as the change from runtime code to compile code, and some others are, such as the matching algorithms optimizations.

In this section, we compare for each refactored matching physical algorithm, i.e., the Sorted Neighborhood Join, Adaptive SNJ, and Traditional Blocking the differences between the old and new implementation in terms of performance. Section 5.3.1 shows the differences for the Sorted Neighborhood Join. For the Adaptive SNJ the results are detailed in Section 5.3.2. We conclude with Section 5.3.3, where we compare the differences between the old and new Traditional Blocking performance.

5.3.1 Sorted Neighborhood Join

The old Sorted Neighborhood Join (SNJ) physical algorithm was a good performer for small datasets, i.e., datasets whose input size is smaller than 5,000 records. However, for bigger datasets, its performance decreased exponentially, as illustrated in Figure 5.12. Moreover, the old algorithm's maximum input size was 100,000 input records. This number compares with the 250,000 input records of the new SNJ physical algorithm. Moreover, the execution times achieved by the new algorithm are very stable. Although the scale does not allow to perceive, the execution time grows at the same factor as the input size. For example, for 5,000 input records, the new SNJ algorithm took 83ms whereas for 25,000 input records, that is, 5 times more records, took 459ms, which is 5.53 times bigger. The average speedup achieved is 68.57, i.e., on average, for datasets with input sizes between 500 and 100,000 input records, the new algorithm is 68.57 times faster.

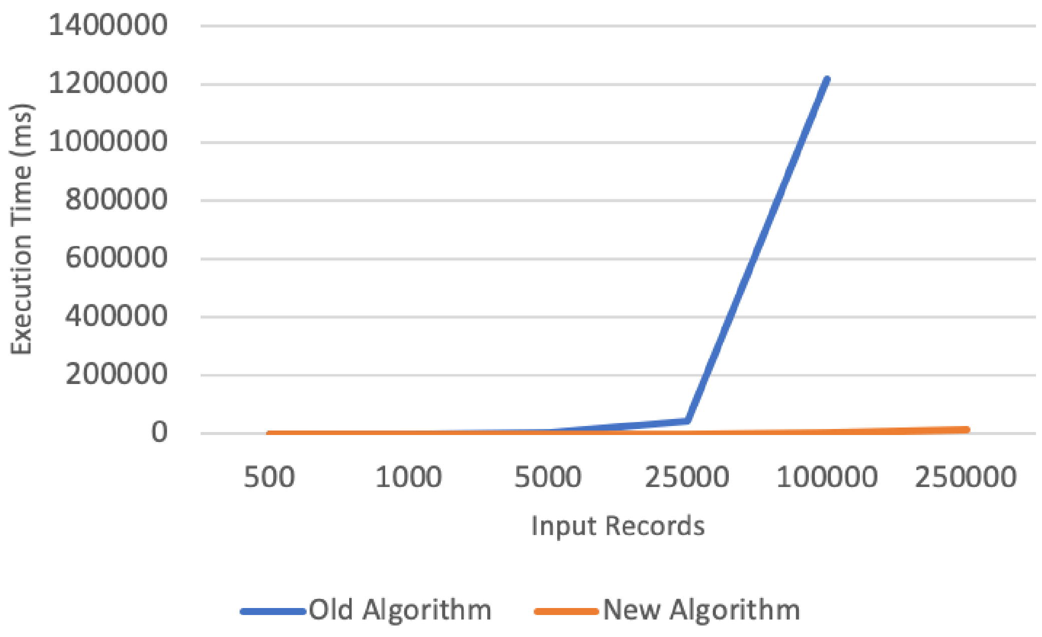
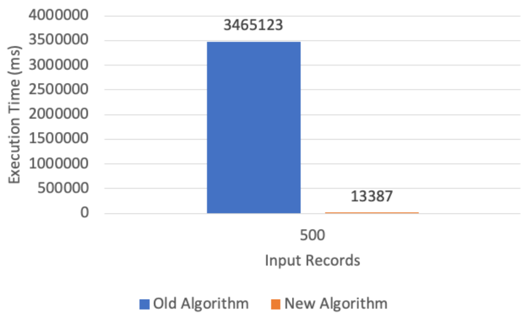


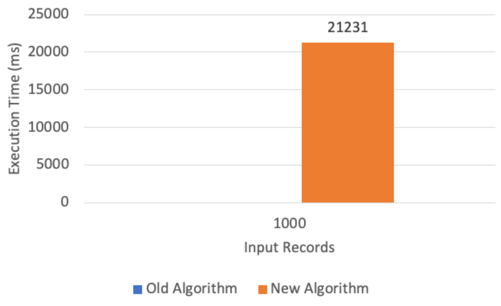
Figure 5.12: Time comparison between the old and new SNJ physical algorithm

5.3.2 Adaptive SNJ

The changes introduced to the Adaptive SNJ are, in their majority, the same as the ones introduced in the algorithm that serves as its base, the Sorted Neighborhood Join. Figure 5.13 shows the time difference between the old and new algorithm for 500 and 1,000 input records, in Figure 5.13a and Figure 5.13b, respectively. For 500 input records, the new algorithm achieves a 258.84 speedup, being this speedup unquantifiable for 1,000 input records since the old algorithm did not support that input size.



(a) Time comparison between the old and new Adaptive SNJ physical algorithm with 500 input records



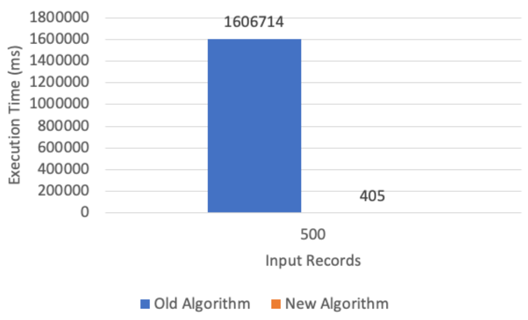
(b) Time comparison between the old and new Adaptive SNJ physical algorithm with 1,000 input records

Figure 5.13: Time comparison between the old and new Adaptive SNJ physical algorithm

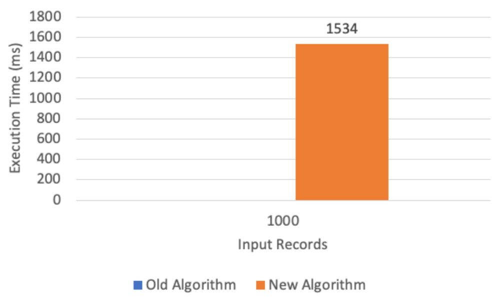
The more memory efficient Map structure enables the refactored physical algorithm to support up to 1,000 input records. Although this algorithm is based on the Sorted Neighborhood Join, it is not able to deal with as much input as that physical algorithm. The main reason is the Adaptive SNJ's dynamic window. Since this window can grow indefinitely, it will demand more comparison than the SNJ. For example, in our experiments, we used a window size of 3 for the SNJ. Therefore, the supporting data structure only needs to store three records, and compare a maximum of six. However, the number of comparisons in the Adaptive SNJ is undetermined. In the worst-case scenario, where all records are identical, it will have a window with the same size as the dataset. If this dataset has 1,000 input records, that would mean performing and storing the result of 1,000,000 comparisons.

5.3.3 Traditional Blocking

The Traditional Blocking physical algorithm refactoring added the possibility to execute this algorithm with 1,000 input records. Moreover, this refactoring achieved a speedup of 3,967.20 times for an input dataset of 500 input records, the maximum input size supported by the old version. In Figure 5.14 we can see the reports of the execution time, in milliseconds, of the old and new traditional blocking physical algorithm for 500 input records (Figure 5.14a), and 1,000 input records (Figure 5.14b).



(a) Time comparison between the old and new Traditional Blocking physical algorithm with 500 input records



(b) Time comparison between the old and new Traditional Blocking physical algorithm with 1,000 input records

Figure 5.14: Time comparison between the old and new Traditional Blocking physical algorithm

Chapter 6

Conclusions

In this document, we detailed the design and integration of an automatic optimizer in CLEENEX. This optimizer is able to automatically decide, for any Data Cleaning Program (DCP) what is the set of physical algorithms that guarantees the best trade-off between effectiveness, i.e., the quality of the results, and performance, i.e., the execution time.

In this document, we focused mainly on the optimization of the matching physical operator. We detailed several matching physical algorithms that enable the scaling up of the default matching algorithm, the Cartesian Product, and discussed the advantages and disadvantages of each one. Some of these matching algorithms achieve better performance by creating less candidate record pairs. However, by doing that, these algorithms may not be able to detect as many approximate duplicates as one that generates more pairs. Moreover, more at an infrastructure level, we described a paradigm change in how a developer can execute and create a matching physical algorithm.

In Section 6.1, we summarize the work done during the creation of the optimizer. We conclude this chapter with Section 6.2, where we detail the future work of this thesis.

6.1 Summary

In Chapter 1, we introduced the concept of *data cleaning tool*, on both its variants, *transformation-based* and *rule-based*. Then, we introduced the *CLEENEX* data cleaning tool, a transformation-based tool, and some of its main features, such as supporting the definition of *Quality Constraints*, i.e., rules that can be applied to the output records of data transformations, and *Manual Data Repairs*, i.e., which allow repairing faulty data that was flagged by a quality constraint that was not satisfied. Moreover, CLEENEX has an architecture with a clear separation between the logical operators, which define what should be executed, and the physical operators, which execute a logical operator, resembling the architecture of a Relational Database Management System (RDBMS). Finally, we referred that the fact that CLEENEX is a research data cleaning tool under development at Instituto Superior Técnino was the main reason to work with that specific data cleaning tool.

In Chapter 2, we detailed how a RDBMS optimizer works. We described several types of optimizers that an RDBMS may use, such as *rule-based optimizer*, *cost-based optimizer*, and a mix between these two. Also in that chapter, we introduced the concept of *data cleaning* and *approximate duplicate detection*.

Chapter 3 is divided into three main sections. In the first one, we detail how we could improve the default approximate duplicate detection algorithm, the Cartesian Product, by using other algorithms that compromise their effectiveness, i.e., the capacity of detecting duplicate records, to achieve better performance. Then, in the second section, we analyzed some research tools that addressed the parallelization and distribution of the approximate duplicate detection task. Finally, in the third section, we described several researching data cleaning tools that address the efficiency of a data cleaning process, where it is included CLEENEX.

In Chapter 4, we detail the implemented optimizer that will be integrated into CLEENEX. In that chapter, we describe how the optimizer integrates into CLEENEX architecture and briefly recapitulate how the various CLEENEX components communicate among them. We also detailed the optimizer architecture and its components. Furthermore, we explained how we perform the paradigm change from runtime code to compile-time code in CLEENEX and some other optimizations that enabled us to achieve better performance in some matching physical algorithms.

In Chapter 5," we perform the experimental validation of our solution. There, we compared how the cost model estimations compare with the real values of the physical operators. Moreover, we also analyzed the performance gain achieved with the optimizations introduced in some matching physical algorithms.

In conclusion, we designed, developed, and integrated an automatic optimizer into CLEENEX that enables it to choose for any Data Cleaning Program the best algorithms possible. This enables CLEENEX to improve its performance and, consequently, its usability.

6.2 Future Work

In order to continue the work of this thesis, we propose the following tasks for future work:

- Single-thread matching algorithms have a scalability problem due to the way they read data. All matching algorithms read the input datasets and store them in memory. This is not scalable. Therefore, one of the most urgent work in CLEENEX is the reimplementation of the matching algorithms so that they are more scalable in a single thread fashion. One possibility, when the algorithms allow, is to read lazily. Another way is to divert to the RDBMS part of the blocking phase. For example, in Traditional Blocking, use the RDBMS to retrieve all records that have the same scale-up key value instead of dividing on the fly as is done at the moment;

- After the single-thread algorithms are with a good enough performance, CLEENEX should support distributed execution. First, we need to assess which data operations can be distributed. Besides the View operator, it should be possible to distribute the remaining operators. The View operator is hard or even impossible to distribute since it is a query to a database. It would be necessary to manipulate the query to distribute the execution. There are two possibilities to distribute the execution: *(i)* create new algorithms that are the distributed version of the current ones, or *(ii)* use tools such as Dedoop and delegate the execution. In what concerns the first option, adding new algorithms is easily achievable due to the architecture of the current implementation. However, it demands reconstructing every algorithm to support a distributed setting, which is very cumbersome. The second option has several limitations. First, we need to find a platform that supports the same operators of CLEENEX. Then, there is also another big problem, the UDFs that are declared and known only by CLEENEX;

- The cost model makes the optimizer possible. One of the goals of every thesis that follows should be the improvement of its accuracy. Namely, one of the first points is the capability to measure the impact of the UDFs in the execution. This impact should be measured both in terms of performance and in the output size of each data operator. There are some options to measure the performance impact, namely: *(i)* static code analysis in conjunction with machine learning, and *(ii)* sampling, i.e., at each execution of a given UDF, save its cost and use it for later estimations. The cost of a UDF will be the average cost of all saved costs.

Bibliography

[1] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi. RHEEM: enabling cross-platform data processing. *PVLDB*, 11(11):1414–1427, 2018. doi: 10.14778/3236187.3236195. URL `http://www.vldb.org/pvldb/vol11/p1414-agrawal.pdf`.

[2] A. N. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI, 8-9 April 2005*, pages 30–39. doi: 10.1109/WIRI.2005.2. URL `https://doi.org/10.1109/WIRI.2005.2`.

[3] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014. doi: 10.14778/2733085.2733096. URL `http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf`.

[4] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD, June 14-16, 2005*, pages 143–154. doi: 10.1145/1066157.1066175. URL `https://doi.org/10.1145/1066157.1066175`.

[5] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Inform.*, 56 (1-2):51–70, 2003. URL `http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-04`.

[6] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE*, 24(9):1537–1555, 2012. doi: 10.1109/TKDE.2011.127. URL `https://doi.org/10.1109/TKDE.2011.127`.

[7] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *SIGKDD, July 23-26, 2002*, pages 475–480. doi: 10.1145/775047.775116. URL `https://doi.org/10.1145/775047.775116`.

[8] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD, June 22-27, 2013*, pages 541–552. doi: 10.1145/2463676.2465327. URL `https://doi.org/10.1145/2463676.2465327`.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[10] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 0124160441, 9780124160446.

[11] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000. URL http://portal.acm.org/citation.cfm?id=377674.377676.

[12] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.

[13] T. Fernandes. A software infrastructure for the cleenex optimizer. Master's thesis, 2015.

[14] H. Galhardas, D. Florescu, D. E. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB, September 11-14, 2001*, pages 371–380, . URL http://www.vldb.org/conf/2001/P371.pdf.

[15] H. Galhardas, A. Lopes, and E. Santos. Support for user involvement in data cleaning. In *DaWaK, August 29-September 2,2011*, pages 136–151, . doi: 10.1007/978-3-642-23544-3_11. URL https://doi.org/10.1007/978-3-642-23544-3_11.

[16] S. Giannakopoulou, M. Karpathiotakis, B. Gaidioz, and A. Ailamaki. Cleanm: An optimizable query language for unified scale-out data cleaning. *PVLDB*, 10(11):1466–1477, 2017. doi: 10.14778/3137628.3137654. URL http://www.vldb.org/pvldb/vol10/p1466-giannakopoulou.pdf.

[17] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[18] G. Graefe and W. J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *ICDE, 1993*, pages 209–218. doi: 10.1109/ICDE.1993.344061.

[19] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *SIGMOD, 1989*, pages 377–388. ISBN 0-89791-317-5. doi: 10.1145/67544.66962. URL http://doi.acm.org/10.1145/67544.66962.

[20] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD, May 22-25, 1995*, pages 127–138. doi: 10.1145/223784.223807. URL https://doi.org/10.1145/223784.223807.

[21] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, Mar. 1996. ISSN 0360-0300. doi: 10.1145/234313.234367. URL http://doi.acm.org/10.1145/234313.234367.

[22] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *SIGMOD, May 31 - June 4, 2015*, pages 1215–1230. doi: 10.1145/2723372.2747646. URL https://doi.org/10.1145/2723372.2747646.

[23] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. R. Borkar, M. J. Carey, and C. Li. Supporting similarity queries in apache asterixdb. In *EDBT 2018*, pages 528–539. doi: 10.5441/002/edbt.2018.64. URL `https://doi.org/10.5441/002/edbt.2018.64`.

[24] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 5(12): 1878–1881, 2012. doi: 10.14778/2367502.2367527. URL `http://vldb.org/pvldb/vol5/p1878_larskolb_vldb2012.pdf`.

[25] S. Kruse, Z. Kaoudi, J. Quiané-Ruiz, S. Chawla, F. Naumann, and B. Contreras. Rheemix in the data jungle - A cross-platform query optimizer -. *CoRR*, abs/1805.03533, 2018. URL `http://arxiv.org/abs/1805.03533`.

[26] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD, August 20-23, 2000*, pages 169–178. doi: 10.1145/347090.347123. URL `https://doi.org/10.1145/347090.347123`.

[27] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014. doi: 10.14778/2732977.2732981. URL `http://www.vldb.org/pvldb/vol7/p1059-dassarma.pdf`.

[28] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Higher Education, 4th edition, 2001. ISBN 0072283637.

[29] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD, June 6-10, 2010*, pages 495–506. doi: 10.1145/1807167.1807222. URL `https://doi.org/10.1145/1807167.1807222`.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1558605703.

[31] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW, April 21-25, 2008*, pages 131–140. doi: 10.1145/1367497.1367516. URL `https://doi.org/10.1145/1367497.1367516`.

[32] S. Yan, D. Lee, M. Kan, and C. L. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *JCDL, June 18-23, 2007*, pages 185–194. doi: 10.1145/1255175.1255213. URL `https://doi.org/10.1145/1255175.1255213`.

[33] S. Yeddula and K. Lakshmaiah. Investigation of techniques for efficient & accurate indexing for scalable record linkage & deduplication. 2012.