# Optimization of Data Cleaning Programs

Tiago Bartolomeu Luiz
Instituto Superior Técnico, Universidade de Lisboa

***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.**

## I. 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 database 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 [1] and BigDansing [2] (research prototypes), most of them they 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).

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).

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, there is a violation that needs to be repaired. One or more 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.

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 as well as data debugging to analyze the source of potential problems affecting data quality. A data cleaning process is represented by a graph of transformations that the user defines. A data transformation is a node of that graph.

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 transformations such as approximate duplicate detection. 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.

This manuscript is organized as follows. Section II provides background about relational optimization and data cleaning concepts. In Section III, we detail techniques to scale-up the approximate duplicate detection, describe research prototypes that distribute the matching execution, and explain some data cleaning research prototypes concerned with the performance of a data cleaning process. The proposed solution is detailed in Section IV. In Section V we evaluate the proposed solution. Finally, Section VI presents the conclusions.

## II. Background

Query optimization is a problem common to all Relational Database Management Systems (RDBMS) in the sense that

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

they all aim at efficiently executing a user's query.

As mentioned in Section I, 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. 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.

### *A. 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 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.

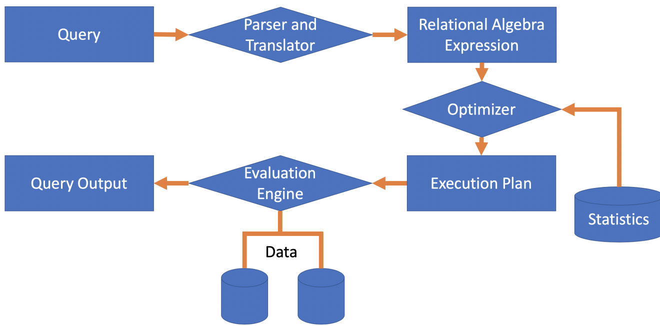


Fig. 1: Relational database query processing steps

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.

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. Most recent optimizers are a combination of both rule-based and cost-based optimizers. These optimizers use first a set of rules to produce an equivalent but cheaper plan and then the optimizer produces exhaustively all equivalent plans using an dynamic programming algorithm.

The query-evaluation plan performance is measured by its cost[3]. The *cost of a plan* is given by the sum of the cost of each node operation.

### *B. 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 are usually solved, or at least minimized, by a data cleaning tool (examples of commercial data cleaning tools are Trifacta[4], Informatica[5], 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 [3] , 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 [2] is an example of a rule-based data cleaning tool.

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 have several millions of tuples, performing a quadratic algorithm is undesirable, and in certain cases, unfeasible.

## III. Related Work

### *A. 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* [4]. 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

[3]Only cost-based optimizers have the notion of a query cost.
[4]https://www.trifacta.com
[5]https://www.informatica.com

crucial step. It is the user's responsibility to define how the key is generated.

*1) Traditional Blocking: Traditional blocking* [5] 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.

*2) Sorted Neighborhood Join: Sorted Neighborhood Join* (SNJ) [6] uses a different approach when compared to Traditional Blocking. SNJ does not create blocks of records, instead, all records are sorted by a predefined key and maintained in their original table (i.e., a single block). 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[6].

Several alternatives to the SNJ were proposed to improve its effectiveness and efficiency. The *Inverted Index Based approach* [6] enables the comparison between records with similar keys 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. The *Adaptive Sorted Neighborhood* [7] overcomes the problem of having a fixed window size. For example, consider that we have a window size of 3 and we have 5 records in a row that are similar. The first and last two records will not be compared since the window size does not allow it. The adaptive sorted neighborhood solves this problem by dynamically changing 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) Canopy Clustering:* The *Canopy Clustering* technique [8] [9] 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 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.

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* [8] [9], and *(ii)* the *Nearest Neighborhood Based Approach* [8] [9]. We explain the first, since it is the most commonly used.

### B. 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, there is the need to perform millions of comparisons when there is large amounts of data. 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, a solution to parallelize and distribute the approximate duplicate detection task, or at least, its most resource demanding subtask, the Cartesian product, is needed.

Figure 2 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.

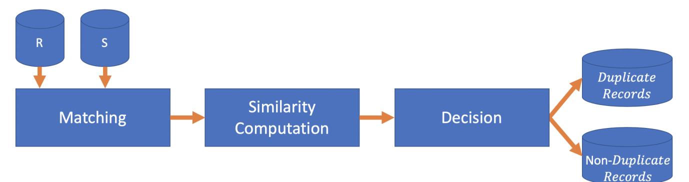


Fig. 2: Approximate duplicate detection task workflow

*1) Dedoop: Dedoop* [10] is a tool to perform efficient deduplication with Hadoop. The underlying idea is to increase performance, since Dedoop implements those techniques in a distributed setting, using the a framework based on the MapReduce (MR) paradigm [11], *Hadoop*. Hadoop uses a cluster of nodes to perform its map and reduce tasks.

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

[6]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.

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.

*2) Parallel Set-Similarity Joins:* Instead of focusing in the deduplication task as a whole, 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[7] joins using MapReduce (MR) was proposed by R. Vernica et al. [12].

The algorithm starts by using the Basic Token Ordering (BTO) [12] algorithm to extract the signatures from the records and compute their occurrence frequency, in relation to the whole dataset. The *signatures*, also known as *partitioning keys*, is a value that is created base on the blocking key value (e.g., from blocking key value "This is very funny" to a signature where the two first words are used, i.e., "this is"). 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) [12], [13] 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 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) [12] 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.

[7]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.

### C. Data Cleaning Research Prototypes

*1) CLEENEX:* CLEENEX [1] 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.

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.

*2) CleanM:* CleanM [3] 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 programFirst, 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* [15]. 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. Then, at the second-level optimization, the comprehensions generated are translated into an intermediate algebra, the *nested relational algebra* [15], 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).

*3) BigDansing:* *BigDansing* [2] is a Big Data Cleansing tool that tackles the problem of efficiency and scalability. *BigDansing* is 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[8].

The *BigDansing* system architecture is illustrated in Figure 3. 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 [16], [17].

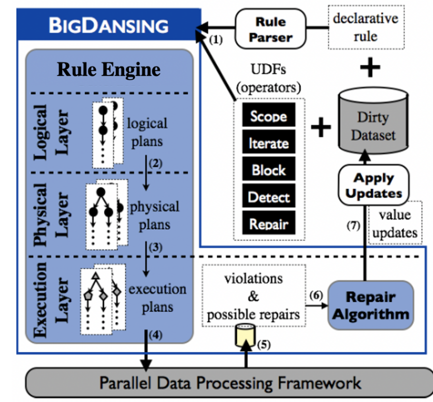


Fig. 3: BigDansing architecture.

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 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.

*4) RHEEM:* *RHEEM* [18] 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

[8] 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.

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.

## IV. Proposed Solution

As stated in Section I, the goal of this thesis is to implement an optimizer to be integrated in the data cleaning research prototype CLEENEX. This optimizer only deals with relational data, since CLEENEX accepts this type of data.

### *A. Optimizer Component Architecture*

The CLEENEX component architecture is represented in Figure 4. It is composed of nine components, including the optimizer.

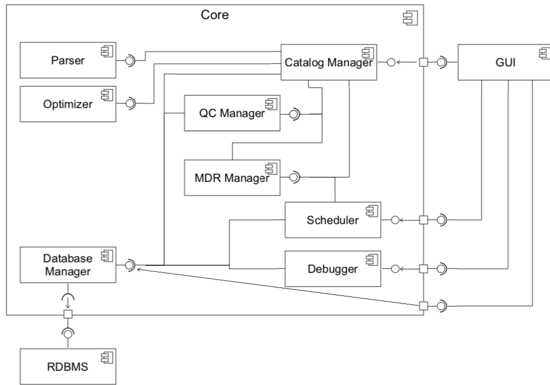


Fig. 4: CLEENEX component architecture

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.

### *B. Optimizer Workflow*

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 5. 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.

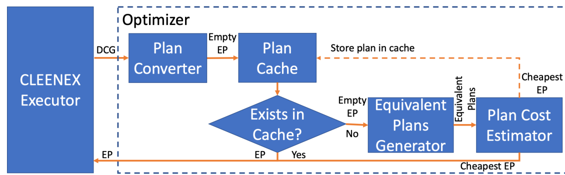


Fig. 5: Optimizer execution workflow

### C. 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 is divided into three main sections: *(i)* the output size estimation, whose formulas for each physical algorithm are shown in Table I*(ii)* the CPU cost measurement, i.e., the cost of a physical algorithm, which allows us to evaluate how much CPU effort is needed to execute the physical algorithm, and *(iii)* the I/O cost measurement, 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. The formulas to perform the measurements of the two last sections are available in Table II.

The cost of a physical algorithm is given by the sum of its estimated I/O and CPU cost multiplied by a penalization factor, i.e., the physical algorithm's cost formula is $(CPU\ \ Cost + I/O\ \ Cost) \times Penalization\ \ Factor$. The penalization factor is a percentage that can vary between 50% and 200%. The factor's goal is to penalize algorithms that are good performers at the cost of having bad quality results, as occurs with the non-default matching algorithms.

This factor is mainly used in the matching physical algorithms, since we needed a way to penalize 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. The penalization factor and cost formula for each physical algorithm is shown in Table III. 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.

To obtain the estimated cost of an execution plan, we need to sum the cost of each of its physical algorithms.

### D. 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 matching physical 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 matching physical algorithms. Although the other physical algorithms also have problems, the matching algorithms' implementation had design choices that impacted severely their performance. Most of the algorithms relied on a List structure to store the generated candidate record pairs. However, this data structure has a search CPU cost of $O(N)$. To decrease significantly the CPU cost, we switched that data structure with a Map, which has $O(1)$ searches.

## V. Evaluation

The experiments enable to validate the solution proposed and described in Section IV. 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. Also we will evaluate the performance gain achieved with the introduction of the optimizer.

### A. Experimental Setup

We compare the performance gain in CLEENEX before and after the introduction of the optimizer. For this, we use multiple variations of the same dataset, the CIDS Publication. This dataset is based in the gold standard dataset CORA [9]. However, the CIDS Publication only contains a subset of CORA dataset, 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 generated from the CIDS publication dataset other six datasets with 500, 1,000, 5,000, 25,000, 100,000, and 250,000 input records.

The experiments were executed in a Macbook Pro 2017 having 4 cores with 2.8GHz and 16 GB of main memory (RAM) of 2133 MHz LPDDR3. The operative system is macOS Catalina version 10.15.6.

The DCP used in our evaluation receives as input one of the datasets that were generated from the CIDS Publication dataset. Then, it has two mapping logical operators, the *AuthorsByPublication*, and the *PubAuthorNames*. They are

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

| 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 | $\frac{N_R \times N_S}{\left(\sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i}\right)^2} \times \sum_{i=1}^{V(BKV_R \cup BKV_S)} \times \frac{1}{i^2}$ [19] | $N_R \times N_S$ |
| | Canopy Clustering | $\frac{N_R \times N_S \times n_l^2}{n_t^2}$ [19] | $N_R \times N_S$ |
| | Sorted Neighborhood Join (SNJ) | $(w^2 + 2(N_R + N_S - w)(w-1))$ [19] | $N_R \times N_S$ |
| | Inverted Index SNJ | $\frac{N_R \times N_S}{V(BKV_R \cup BKV_S)^2}(w^2 + (V(BKV_R \cup BKV_S) - w)(2w-1))$ [19] | $N_R \times N_S$ |
| | 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 I: CLEENEX output size estimation. $N_R$ 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 $S$.

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

TABLE II: 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 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 $N_{R \cup S}$

followed by a matching logical operator that receives as input the last mapping logical operator output, *PubAuthorNames*. The matching is a self-matching, i.e., the input table is read twice. Then, the DCP finished with a clustering logical operator, *ClusterAuthors*, followed by a merging logical operator, *CleanAuthors*.

### B. Metrics

To evaluate the developed cost model, we used the the error rate metric, which enables to evaluate the difference between the real output size and the estimated output size. The error rate has two variants, one for non-matching physical algorithms (Equation 1), and other for matching physical algorithms (Equation 2). The reason why the matching physical algorithms have different formulas is because the cost model for those algorithms measures the number of candidate pairs generated instead of the matching physical operator output, which includes the filtering phase, i.e., the application of the $WHERE$ clause defined in the matching logical operator. For both variants, an error rate less than 1.0 means that the estimated value is higher than the real output, greater than 1.0 the estimated value is smaller than the real output, and when the error rate is 1.0 the estimated value and real output are the same.

$$Error\ Rate = \frac{Physical\ Operator\ Real\ Output}{Estimated\ Output} \tag{1}$$

$$Error\ Rate_{Matching} = \frac{Candidate\ Pairs\ Generated}{Estimated\ Pairs\ Generated} \tag{2}$$

### C. 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. In this evaluation, we focused on the output size estimation since both the CPU and I/O cost are computed based on it. More precisely, in this document we cover the cost model results for matching operator. The mapping physical operators that precede the matching, have an error rate of 3.37. Therefore, for 1000 input

| Data Operator | 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 | 199,96% | 0,01% $\rightarrow$ 50,00% | 124,98% | $CPU + I/O \times 124,98\%$ |
| | Traditional Blocking | 194,39% | 0,35% $\rightarrow$ 50,00% | 122,20% | $CPU + I/O \times 122,20\%$ |
| | Adaptive SNJ | 194,39% | 8,47% $\rightarrow$ 50,00% | 122,20% | $CPU + I/O \times 122,20\%$ |
| | Inverted Index SNJ | 185,61% | 5013,83 $\rightarrow$ 200,00% | 192,81% | $CPU + I/O \times 192,81\%$ |
| | Canopy Clustering | 187,67% | 1619,64% $\rightarrow$ 200,00% | 193,84% | $CPU + I/O \times 193,84\%$ |

TABLE III: Algorithm's cost formulas with penalization factor

records, the cost model estimates that there are 1000 output records, but instead, there were 3389.

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.

For the Sorted Neighborhood Join, Figure 6 shows that the cost model is consistently accurate across the various input sizes tested. As shown in Table IV, 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.

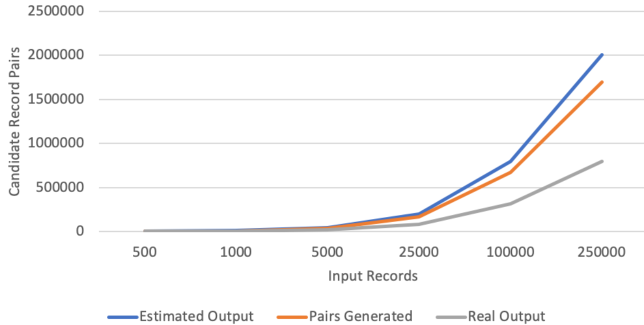


Fig. 6: Cost model results for *SimilarAuthors* (Matching) with Sorted Neighborhood Join

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 7 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 V. Therefore, the formula for the output size estimation, extracted from [19], is not able to precisely estimate for the Inverted Index SNJ. Alternatively, the authors of [19] 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.

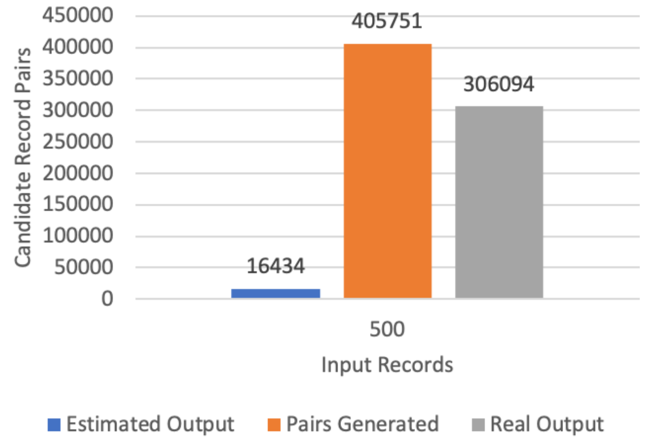


Fig. 7: Cost model results for *Similar Authors* (Matching) with Inverted Index SNJ

The cost model results for the Adaptive SNJ are illustrated in Figure 8. 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 VI 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 8, 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

| | 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 IV: Cost model results summary for *SimilarAuthors* (Matching) with Sorted Neighborhood Join

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

TABLE V: Cost model results summary for *SimilarAuthors* (Matching) with Inverted Index SNJ

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 matching logical operator, 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 VI, 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.

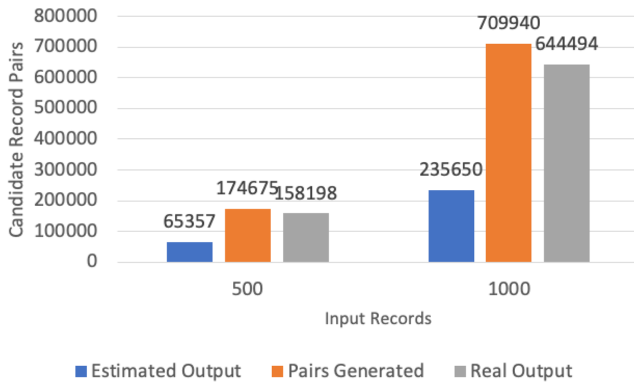


Fig. 8: Cost model results for *SimilarAuthors* (Matching) with Adaptive SNJ

| | 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 VI: Cost model results summary for *SimilarAuthors* (Matching) with Adaptive SNJ

Both Adaptive SNJ and Traditional Blocking share their output estimation formulas. Moreover, these formulas assume that the input dataset uses a *Zipf Distribution* [20]. 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 8 and Table VI 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 9. On average, the error rate with the normal distribution is 317.91, i.e., almost 112 times bigger than with the Zipf distribution.

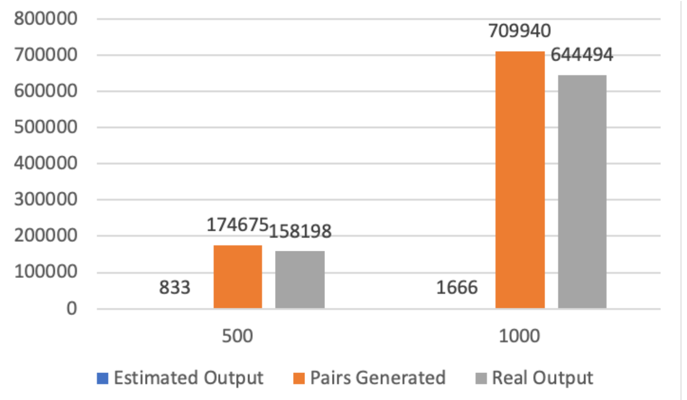


Fig. 9: Cost model results for *Similar Authors* (Matching) with Adaptive SNJ using Normal Distribution

The results for the Traditional Blocking's cost model are identical to the Adaptive SNJ's, reported in Table VI. 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 10. 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 VII.

| | 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 VII: 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

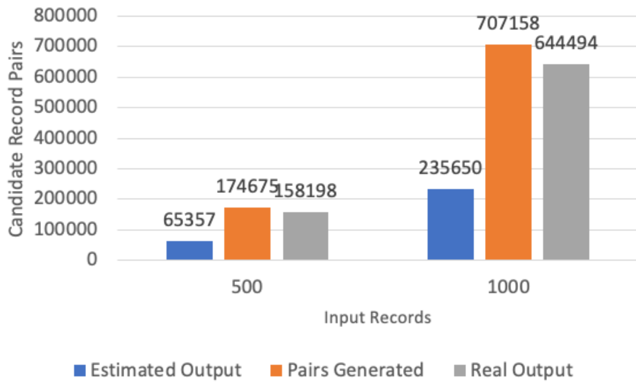


Fig. 10: Cost model results for *SimilarAuthors* (Matching) with Traditional Blocking

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 11, the estimation of generated candidate record pairs is not 100% accurate. As Table VIII 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, which is the real output size of the preceding matching physical operator. 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.

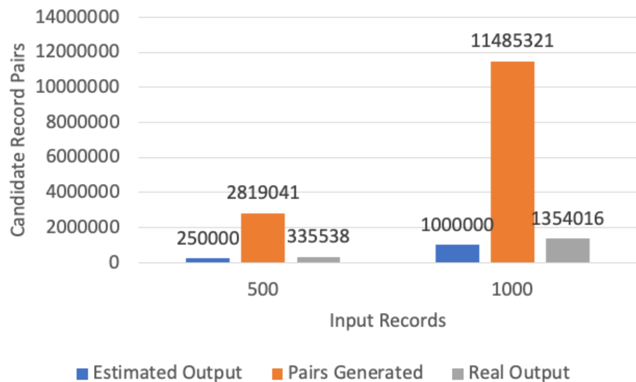


Fig. 11: Cost model results for *SimilarAuthors* (Matching) with Cartesian Product

| | 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 VIII: 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 12, the estimations made are only 1.63 times smaller than the real values, i.e., the error rate, as reported in Table IX, is just 1.63.

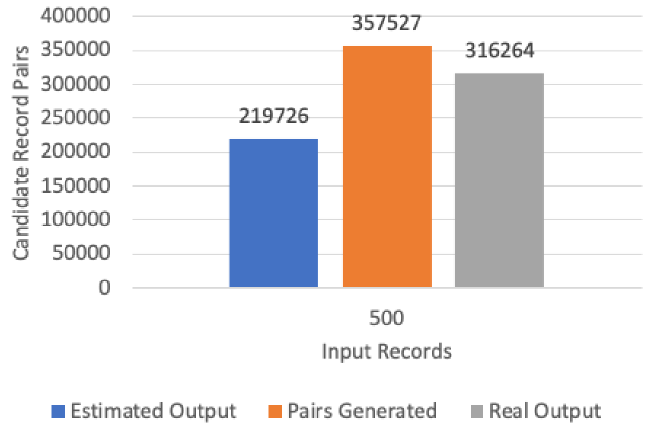


Fig. 12: Cost model results for *SimilarAuthors* (Matching) with Canopy Clustering

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

TABLE IX: Cost model results summary for *SimilarAuthors* (Matching) with Canopy Clustering

## VI. Conclusion

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.

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. Moreover, the optimizations performed in the Traditional Blocking, the Sorted Neighborhood Join, and Adaptive SNJ physical algorithms achieved execution times that are, on average, 1,430.87 faster than the execution times previous to the optimizations.

## REFERENCES


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

[2] 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*, pp. 1215–1230. [Online]. Available: https://doi.org/10.1145/2723372.2747646

[3] S. Giannakopoulou, M. Karpathiotakis, B. Gaidioz, and A. Ailamaki, "Cleanm: An optimizable query language for unified scale-out data cleaning," *PVLDB*, vol. 10, no. 11, pp. 1466–1477, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1466-giannakopoulou.pdf

[4] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

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

[6] M. A. Hernández and S. J. Stolfo, "The merge/purge problem for large databases," in *SIGMOD, May 22-25, 1995*, pp. 127–138. [Online]. Available: https://doi.org/10.1145/223784.223807

[7] S. Yan, D. Lee, M. Kan, and C. L. Giles, "Adaptive sorted neighborhood methods for efficient record linkage," in *JCDL, June 18-23, 2007*, pp. 185–194. [Online]. Available: https://doi.org/10.1145/1255175.1255213

[8] 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*, pp. 475–480. [Online]. Available: https://doi.org/10.1145/775047.775116

[9] 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*, pp. 169–178. [Online]. Available: https://doi.org/10.1145/347090.347123

[10] L. Kolb, A. Thor, and E. Rahm, "Dedoop: Efficient deduplication with hadoop," *PVLDB*, vol. 5, no. 12, pp. 1878–1881, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1878_larskolb_vldb2012.pdf

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[12] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD, June 6-10, 2010*, pp. 495–506. [Online]. Available: https://doi.org/10.1145/1807167.1807222

[13] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW, April 21-25, 2008*, pp. 131–140. [Online]. Available: https://doi.org/10.1145/1367497.1367516

[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*, pp. 371–380. [Online]. Available: http://www.vldb.org/conf/2001/P371.pdf

[15] L. Fegaras and D. Maier, "Optimizing object queries using an effective calculus," *ACM Trans. Database Syst.*, vol. 25, no. 4, pp. 457–516, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=377674.377676

[16] 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*, pp. 143–154. [Online]. Available: https://doi.org/10.1145/1066157.1066175

[17] 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*, pp. 541–552. [Online]. Available: https://doi.org/10.1145/2463676.2465327

[18] 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*, vol. 11, no. 11, pp. 1414–1427, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1414-agrawal.pdf

[19] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *IEEE*, vol. 24, no. 9, pp. 1537–1555, 2012. [Online]. Available: https://doi.org/10.1109/TKDE.2011.127

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