Scaling Big Data Cleansing

Dissertation by
Zuhair Yarub Khayyat

In Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy

King Abdullah University of Science and Technology
Thuwal, Kingdom of Saudi Arabia

May, 2017
The dissertation of Zuhair Yarub Khayyat is approved by the examination committee.

Committee Chairperson: Panos Kalnis — Professor
Committee Members: Mohamed-Slim Alouini — Professor, Basem Shihada — Associate Professor, Marco Canini — Assistant Professor, Christoph Koch — Professor
ABSTRACT

Scaling Big Data Cleansing
Zuhair Yarub Khayyat

Data cleansing approaches have usually focused on detecting and fixing errors with little attention to big data scaling. This presents a serious impediment since identifying and repairing dirty data often involves processing huge input datasets, handling sophisticated error discovery approaches and managing huge arbitrary errors. With large datasets, error detection becomes overly expensive and complicated especially when considering user-defined functions. Furthermore, a distinctive algorithm is desired to optimize inequality joins in sophisticated error discovery rather than naively parallelizing them. Also, when repairing large errors, their skewed distribution may obstruct effective error repairs. In this dissertation, I present solutions to overcome the above three problems in scaling data cleansing.

First, I present BigDansing as a general system to tackle efficiency, scalability, and ease-of-use issues in data cleansing for Big Data. It automatically parallelizes the user’s code on top of general-purpose distributed platforms. Its programming interface allows users to express data quality rules independently from the requirements of parallel and distributed environments. Without sacrificing their quality, BigDansing also enables parallel execution of serial repair algorithms by exploiting the graph representation of discovered errors. The experimental results show that BigDansing outperforms existing baselines up to more than two orders of magnitude.

Although BigDansing scales cleansing jobs, it still lacks the ability to handle sophisticated error discovery requiring inequality joins. Therefore, I developed IEJoin as an algorithm for fast inequality joins. It is based on sorted arrays and space efficient
bit-arrays to reduce the problem’s search space. By comparing IEJoin against well-known optimizations, I show that it is more scalable, and several orders of magnitude faster.

**BIGDANSING** depends on vertex-centric graph systems, *i.e.*, Pregel, to efficiently store and process discovered errors. Although Pregel scales general-purpose graph computations, it is not able to handle skewed workloads efficiently. Therefore, I introduce **Mizan**, a Pregel system that balances the workload transparently during runtime to adapt for changes in computing needs. **Mizan** is general; it does not assume any a priori knowledge of the graph structure or the algorithm behavior. Through extensive evaluations, I show that **Mizan** provides up to 84% improvement over techniques leveraging static graph pre-partitioning.
ACKNOWLEDGEMENTS

To my parents, Yarub and Naida. I simply cannot thank you enough; I would have never been able to succeed in my Ph.D. without your unconditional love, encouragement and faith. I will forever be indebted for your efforts in raising me and the education you provided me. I still remember my first programming experience Logo in from the 90s, which was the cornerstone for my passion in Computer Science.

To my charming wife, Manal. I am grateful for sharing my journey toward my Ph.D.; it has been a unique experience for both of us. I am eternally grateful for your continuous love and support during this rough period. Being my wife is the best thing that ever happened to me while at KAUST.

To my lovely kids, Mazin and Ziyad. Your enthusiasm in learning and doing new things always inspires me. Your energy has been consistently feeding my determination and keeping me up all night. I am thankful for being in my life; my Ph.D. journey would have never been exciting without you.

To my advisor, Panos Kalnis. Thank you for your confidence in me and your exceptional support throughout my Ph.D. With your teachings, I was able to compete with students from top universities and publish my work in high-quality venues.

To my mentor, Hani Jamjoom. Thank you for your patience and valuable guidance in my early Ph.D. years. Your knowledge and experience in scientific writing helped me throughout my whole Ph.D.

To my academic family, InfoCloud members. Thank you for your good company, positive influence, and support through my rough times.

Portions of the research in this dissertation used the MDC Database made available by Idiap Research Institute, Switzerland and owned by Nokia.
TABLE OF CONTENTS

Examination Committee Page | 2
Copyright | 3
Abstract | 4
Acknowledgements | 6
Table of Contents | 7
List of Figures | 11
List of Tables | 15

1 Introduction | 17
  1.1 Dealing with Big Datasets | 19
  1.2 Sophisticated Cleaning Jobs | 21
  1.3 Managing Large Random Violation Graphs | 23
  1.4 Contributions and dissertation Organization | 25

2 Related Work | 28
  2.1 Data Cleaning | 28
  2.2 Scaling Inequality Joins | 31
  2.3 General-purpose Scalable Graph Processing | 34

3 BigDansing: A System for Big Data Cleansing | 43
  3.1 Fundamentals and An Overview | 43
    3.1.1 Data Cleansing Semantics | 44
    3.1.2 Architecture | 46
  3.2 Rule Specification | 49
    3.2.1 Logical Operators | 49
    3.2.2 BigDansing Example Job | 54
    3.2.3 The Planning Process | 54
3.3 Building Physical Plans ........................................................................... 56
  3.3.1 Physical Operators ........................................................................... 57
  3.3.2 From Logical to Physical Plan ............................................................... 58
  3.3.3 Fast Joins with Ordering Comparisons ................................................ 63
  3.3.4 Translation to Spark Execution Plans .................................................. 65
  3.3.5 Translation to MapReduce Execution Plans ........................................ 67
  3.3.6 Physical Data Access ......................................................................... 68
3.4 Distributed Repair Algorithms ................................................................. 69
  3.4.1 Scaling Data Repair as a Black Box ..................................................... 70
  3.4.2 Scalable Equivalence Class Algorithm ................................................. 73
3.5 Experimental Study .............................................................................. 74
  3.5.1 Setup .................................................................................................. 74
  3.5.2 Single-Node Experiments .................................................................. 77
  3.5.3 Multi-Node Experiments ................................................................... 80
  3.5.4 Scaling BIGDANSING Out ................................................................ 82
  3.5.5 Deduplication in BIGDANSING ......................................................... 82
  3.5.6 BIGDANSING In-Depth .................................................................... 83
3.6 Use Case: Violation Detection in RDF ..................................................... 86
  3.6.1 Source of Errors .............................................................................. 86
  3.6.2 Integrity Violation Example ............................................................... 87
  3.6.3 Violation Detection Plan in BIGDANSING ......................................... 88
  3.6.4 Experiments .................................................................................... 91
4 Fast and Scalable Inequality Joins .............................................................. 93
  4.1 Solution Overview .............................................................................. 93
  4.2 Centralized Algorithms ....................................................................... 97
    4.2.1 IEJoin ............................................................................................ 98
    4.2.2 IESelfJoin .................................................................................... 102
    4.2.3 Enhancements .............................................................................. 104
  4.3 Query Optimization ............................................................................. 106
    4.3.1 Selectivity Estimation ................................................................... 107
    4.3.2 Join Optimization with Multiple Predicates .................................... 108
    4.3.3 Multi-way Join Optimization ......................................................... 109
  4.4 Incremental Inequality Joins .................................................................. 110
  4.5 Scalable Inequality Joins ...................................................................... 114
    4.5.1 Scalable IEJoin ............................................................................ 114
4.5.2 Multithreaded and Distributed IEJoin ........................................... 117
4.6 Integration into Existing Systems ..................................................... 118
  4.6.1 PostgreSQL ............................................................................. 118
  4.6.2 Spark SQL ............................................................................... 120
  4.6.3 BigDANSING .......................................................................... 123
  4.6.4 Rheem ..................................................................................... 124
4.7 Experimental Study ........................................................................ 125
  4.7.1 Datasets .................................................................................. 125
  4.7.2 Algorithms .............................................................................. 126
  4.7.3 Queries ................................................................................... 128
  4.7.4 Setup ...................................................................................... 131
  4.7.5 Parameter Setting .................................................................... 131
  4.7.6 Single-node Experiments .......................................................... 134
  4.7.7 IEJoin Optimizer Experiments ................................................... 139
  4.7.8 Incremental IEJoin Experiments ............................................... 143
  4.7.9 Multi-node Experiments ............................................................ 146

5 Mizan: A System for Dynamic Load Balancing in Large-scale Graph
  Processing ......................................................................................... 152
  5.1 Dynamic Behavior of Algorithms .................................................. 152
    5.1.1 Example Algorithms ............................................................... 156
  5.2 Mizan .......................................................................................... 158
    5.2.1 Monitoring ............................................................................. 159
    5.2.2 Migration Planning ................................................................. 160
  5.3 Implementation ............................................................................. 165
    5.3.1 Vertex Ownership .................................................................. 167
    5.3.2 DHT Updates After Vertex Migration ...................................... 168
    5.3.3 Migrating Vertices with Large Message Size ......................... 169
  5.4 Evaluation .................................................................................... 170
    5.4.1 Giraph vs. Mizan ................................................................. 171
    5.4.2 Effectiveness of Dynamic Vertex Migration ............................ 173
    5.4.3 Overhead of Vertex Migration .............................................. 178
    5.4.4 Scalability of Mizan ............................................................. 179

6 Concluding Remarks ........................................................................ 182
  6.1 Summary of Contributions ........................................................... 182
  6.2 Future Research Directions .......................................................... 184
LIST OF FIGURES

1.1 The violation graph of $\phi_F$ and $\phi_D$. Undirected edges represent commutative violations, while edge labels represent the violated rule and its attributes. .................................................. 19
1.2 East-Coast and West-Coast transactions .................................................. 22

2.1 Example of hash-based partitioning with three partitions and seven edge cuts .................................................. 36
2.2 Example of three-way min-cuts partitioning using METIS with four edge cuts .................................................. 37

3.1 BigDANSING architecture ..................................................................... 46
3.2 Logical operators execution for rule FD $\phi_F$ .................................................. 52
3.3 Planner execution flow ........................................................................ 55
3.4 Example of a logical plan ..................................................................... 56
3.5 Plans for DC in rule 3.1 ........................................................................ 60
3.6 Example plans with CoBlock .................................................................. 62
3.7 A Summary of BigDANSING’s logical, physical and execution operators for Spark .................................................. 65
3.8 Data repair as a black box ..................................................................... 70
3.9 Data cleansing times ........................................................................ 77
3.10 Single-node experiments with $\varphi_1$, $\varphi_2$, and $\varphi_3$ .................................................. 78
3.11 Multi-nodes experiments with $\varphi_1$, $\varphi_2$ and $\varphi_3$ .................................................. 79
3.12 Results for the experiments on (a) scale-out, (b) deduplication, and (c) physical optimizations .................................................. 81
3.13 (a) Abstraction and (b) scalable repair .................................................. 83
3.14 An example RDF graph ..................................................................... 87
3.15 An integrity constraint query graph in RDF .................................................. 89
3.16 The logical plan of the UDF detection rule in Figure 3.15 .................................................. 89
3.17 Data transformations within the logical operators in Figure 3.16 .................................................. 90
3.18 Violation detection performance on dirty RDF data .................................................. 91
4.1 IEJoin process for query $Q_p$ .............................................. 96
4.2 IEJoin process for query $Q_t$ .............................................. 99
4.3 Example of using a bitmap .................................................. 105
4.4 Incremental IESelfJoin process for query $Q_p$ ......................... 113
4.5 An example of the pre-processing stage with two relations $R$ and $S$ 117
4.6 Spark’s DAG for the pre-processing phase .................................. 122
4.7 Spark’s DAG for IEJoin the post-processing phase ....................... 123
4.8 A Summary of the modified BIGDANSING’s logical, physical and execution operators for Spark with IEJoin. ......................... 124
4.9 IEJoin vs baseline systems (centralized) .................................. 135
4.10 IEJoin vs. BTree and GiST (centralized) ................................. 136
4.11 $Q_5$ runtime with different Female distributions, and $Q_5$, $Q_6$ and $Q_7$ runtimes with 10% Female distribution ........................................... 138
4.12 $Q_1$ runtime for data that fits caches .................................... 138
4.13 Result size and runtime for $Q_1'$ ........................................... 140
4.14 Runtime of $\Delta Inc$, $B-\Delta Inc$, non-incremental, and $P-\Delta Inc$ on small update sizes .................................................. 144
4.15 Runtime and output size of non-incremental and $B-\Delta Inc$ IEJoin on big update sizes .................................................. 145
4.16 Memory and I/O overhead of $B-\Delta Inc$, and runtime and output size of $B-\Delta Inc$ and $\sigma Inc$ (30M cached input) ......................... 147
4.17 Distributed IEJoin (100M tuples, 6 nodes/threads) ....................... 148
4.18 Distributed IEJoin, 6B tuples, 16 nodes .................................. 148
4.19 Runtime of IEJoin ($c = 10$) .............................................. 149
4.20 Without result materialization ($c = 5$, 10) ................................. 150
4.21 Speedup (8B rows) & Scaleup on $Q_1$ .................................... 150

5.1 Factors that can affect the runtime in the Pregel framework .......... 153
5.2 The difference in execution time when processing PageRank on different graphs using hash-based, range-based and min-cuts partitioning.
Because of its size, arabic-2005 cannot be partitioned using min-cuts (ParMETIS) in the local cluster ........................................... 155
5.3 The difference between stationary and non-stationary graph algorithms with respect to the incoming messages. $Total$ represents the sum across all workers and $Max$ represents the maximum amount (on a single worker) across all workers. ........................................... 156
5.4 The statistics monitored by MIZAN ........................................ 159
5.5 MIZAN’s BSP flow with migration planning: ............................... 160
5.6 Summary of MIZAN’s Migration Planner .................................. 162
5.7 Matching senders (workers with vertices to migrate) with receivers using their summary statistics .................................. 165
5.8 Architecture of MIZAN in light of the proposed graph processing system .................................................. 167
5.9 Migrating vertex $v_i$ from Worker 1 to Worker 2, while updating its DHT home worker (Worker 3) ............................................ 168
5.10 Delayed vertex migration .......................................................... 169
5.11 Comparing Static MIZAN vs. Giraph using PageRank on social network and random graphs ............................................. 172
5.12 Comparing MIZAN vs. Giraph using PageRank on regular random graphs, the graphs are uniformly distributed with each has around 17M edge ................................................................. 173
5.13 Comparing Static MIZAN and Work Stealing (Pregel clone) vs. MIZAN using PageRank on a social graph (LiveJournal1). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph. .................................................. 173
5.14 Comparing MIZAN’s migration cost with the algorithm runtime at each superstep using PageRank on a social graph (LiveJournal1) starting with range based partitioning. The points on the superstep’s runtime trend represents MIZAN’s migration objective at that specific superstep. ................................................................. 174
5.15 Comparing both the total migrated vertices and the maximum migrated vertices by a single worker for PageRank on LiveJournal1 starting with range-based partitioning. The migration cost at each superstep is also shown .................................................. 175
5.16 Comparing static MIZAN and Work Stealing (Pregel clone) vs. MIZAN using Top-K PageRanks on a social graph (LiveJournal1). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph .................................................. 176
5.17 Comparing Work stealing (Pregel clone) vs. MIZAN using DMST and advertisement propagation Simulation on a metis partitioned social graph (LiveJournal1) .................................................. 177
5.18 Comparing both the total migrated vertices and the maximum migrated vertices by a single worker for advertisement propagation on LiveJournal1 starting with METIS partitioning. The migration cost at each superstep is also shown. 178

5.19 Comparing static Mizan and Work Stealing (Pregel clone) vs. Mizan using advertisement propagation on a social graph (LiveJournal1) starting with METIS partitioning. The points on the Mizan’s trend represents Mizan’s migration objective at that specific superstep. 178

5.20 Speedup on a 16 machines Linux Cluster using PageRank on the hollywood-2011 dataset. 180

5.21 Speedup on the IBM Blue Gene/P supercomputer using PageRank on the arabic-2005 dataset. 181

A.1 Example of bushy plan 200
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Dataset $D$ with tax data records</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Statistics of the datasets</td>
<td>75</td>
</tr>
<tr>
<td>3.2</td>
<td>Integrity constraints used for testing</td>
<td>75</td>
</tr>
<tr>
<td>3.3</td>
<td>Repair quality using HAI and TaxB datasets</td>
<td>85</td>
</tr>
<tr>
<td>4.1</td>
<td>Secondary key sorting order for $Y/X$ in $\text{op}_1/\text{op}_2$</td>
<td>104</td>
</tr>
<tr>
<td>4.2</td>
<td>Permutation array creation for self-join $Q_p$</td>
<td>120</td>
</tr>
<tr>
<td>4.3</td>
<td>Size of the datasets</td>
<td>125</td>
</tr>
<tr>
<td>4.4</td>
<td>Bitmaps on 10M tuples (Events data)</td>
<td>131</td>
</tr>
<tr>
<td>4.5</td>
<td>Bitmaps on 10M tuples with max scan index optimization (Events data)</td>
<td>132</td>
</tr>
<tr>
<td>4.6</td>
<td>Cache statistics on 10M tuples (Events data)</td>
<td>133</td>
</tr>
<tr>
<td>4.7</td>
<td>IEJoin time analysis on 100M tuples and 6 workers</td>
<td>134</td>
</tr>
<tr>
<td>4.8</td>
<td>Runtime and memory usage (PG-IEJOIN)</td>
<td>137</td>
</tr>
<tr>
<td>4.9</td>
<td>Time breakdown on 50M tuples, all times in seconds</td>
<td>137</td>
</tr>
<tr>
<td>4.10</td>
<td>$Q_1$'s selectivity estimation on 50M tuples Employees dataset with low selectivity $\text{age}$ attribute</td>
<td>139</td>
</tr>
<tr>
<td>4.11</td>
<td>$Q_1$'s selectivity estimation on 50M tuples Employees dataset with high selectivity $\text{age}$ attribute</td>
<td>139</td>
</tr>
<tr>
<td>4.12</td>
<td>Relation sizes, selectivity estimations and actual output for individual joins in $Q_{mw}$</td>
<td>142</td>
</tr>
<tr>
<td>4.13</td>
<td>Runtime of different multi-way join plans for $Q_{mw}$</td>
<td>142</td>
</tr>
<tr>
<td>5.1</td>
<td>Datasets—$N$, $E$ denote nodes and edges, respectively. Graphs with prefix $kg$ are synthetic</td>
<td>170</td>
</tr>
<tr>
<td>5.2</td>
<td>Overhead of MIZAN’s migration process when compared to the total runtime using range partitioning with PageRank, and METIS partitioning with advertisement propagation Simulation on LiveJournal1 graph</td>
<td>179</td>
</tr>
</tbody>
</table>
5.3 Scalability of MIZAN on a 16 machines Linux Cluster (hollywood-2011 dataset), and an the IBM Blue Gene/P supercomputer (arabic-2005 dataset).
Chapter 1

Introduction

Data quality is a major concern in terms of its scale, as more than 25% of critical data of the world’s top companies is flawed [1]. In fact, cleansing dirty data is a fundamental challenge that has been studied for decades [2]. Nevertheless, data quality is gaining lots of attention in the era of Big Data due to its increasing complexity. It has been reported that data scientists are spending 60% of their time cleaning and organizing data [3]. This is because data cleansing is hard; data errors arise in different forms, i.e., typos, duplicates, non-compliance with business rules, outdated data, and missing values.

Example 1: Table 1.1 shows a sample tax data $D$ in which each record represents an individual’s information. Suppose that the following three data quality rules need to hold on $D$: (r1) a zipcode uniquely determines a city; (r2) given two distinct individuals, the one earning a lower salary should have a lower tax rate; and (r3) two tuples refer to the same individual if they have similar names, and their cities are inside the same county. I define these rules as follows:

$r1$ \( \phi_F : D(\text{zipcode} \rightarrow \text{city}) \)

$r2$ \( \phi_D : \forall t_i, t_j \in D, \neg(t_i.\text{salary} < t_j.\text{salary} \wedge t_i.\text{rate} > t_j.\text{rate}) \)

$r3$ \( \phi_U : \forall t_i, t_j \in D, \neg(\text{simF}(t_i.\text{name}, t_j.\text{name}) \wedge \text{getCounty}(t_i.\text{city}) = \text{getCounty}(t_j.\text{city})) \)

Note that $\phi_F$, $\phi_D$, and $\phi_U$ can respectively be expressed as a functional dependency (FD), a denial constraint (DC), and a user-defined function (UDF) by using procedural language. $\phi_U$ requires an ad-hoc similarity function and access to a mapping table to
Table 1.1: Dataset $D$ with tax data records

<table>
<thead>
<tr>
<th>name</th>
<th>zipcode</th>
<th>city</th>
<th>state</th>
<th>salary</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Annie</td>
<td>10001</td>
<td>NY</td>
<td>NY</td>
<td>24000</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Laure</td>
<td>90210</td>
<td>LA</td>
<td>CA</td>
<td>25000</td>
</tr>
<tr>
<td>$t_3$</td>
<td>John</td>
<td>60601</td>
<td>CH</td>
<td>IL</td>
<td>40000</td>
</tr>
<tr>
<td>$t_4$</td>
<td>Mark</td>
<td>90210</td>
<td>SF</td>
<td>CA</td>
<td>88000</td>
</tr>
<tr>
<td>$t_5$</td>
<td>Robert</td>
<td>60827</td>
<td>CH</td>
<td>IL</td>
<td>15000</td>
</tr>
<tr>
<td>$t_6$</td>
<td>Mary</td>
<td>90210</td>
<td>LA</td>
<td>CA</td>
<td>81000</td>
</tr>
</tbody>
</table>

obtain the county information. Tuples $t_2$ and $t_4$ form an error $w.r.t.$ $\phi_F$, as well as $t_4$ and $t_6$, because they have the same zipcode but different city values. Tuples $t_1$ and $t_2$ violate $\phi_D$, because $t_1$ has a lower salary than $t_2$ but pays a higher tax; so do $t_5$ and $t_2$. Rule $\phi_U$ does not have data errors, i.e., no duplicates in $D$ $w.r.t.$ $\phi_U$. \qed

Cleansing the above dataset typically consists of three steps: (1) specifying quality rules; (2) detecting data errors $w.r.t.$ the stated rules; and (3) repairing detected errors by removing records, modifying their content or by adding new records. Generally speaking, after step (1), the data cleansing process iteratively runs steps (2) and (3) until obtaining an instance of the data (a.k.a. a repair) that satisfies the specified rules. These three steps have been extensively studied, by both academia and industry, in single-node settings \cite{4-11}. However, the primary focus has been on how to repair errors with the highest accuracy, without addressing the issue of efficient and scalable detection and repair. With big data, three scaling problems arise from processing the above rules ($\phi_F$, $\phi_D$, and $\phi_U$), and cleaning their errors. The first issue is the magnitude of the data, where the detection process becomes too expensive when enumerating all pairs of tuples with massive data. Furthermore, the size of discovered errors grows with larger datasets, which results in a huge processing overhead for the user’s serial repair algorithms. It is not practical to implement a scalable code for every error identification technique. Similarly, it might not be possible to parallelize repair algorithms due to their complexity and strict quality guarantees. The second problem is provoked from complex error discovery approaches that require joins based
on inequality conditions, i.e., \((t_i \text{.salary} < t_j \text{.salary})\) and \((t_i \text{.rate} > t_j \text{.rate})\) in \(\phi_D\). This type of joins is extremely expensive even for small datasets; no system or algorithm in the literature can process it efficiently. The third problem is related to the complexity of storing, handling and pre-processing discovered errors (a.k.a. violations) before passing them to the repair algorithm. Many researchers use a graph representation of violations (a.k.a. violation graph) to store the rich content of discovered errors, improve the user’s perception of the invalid relationships in the data, and to achieve optimal repair decisions. In Figure 1.1, I show the violation graph when applying rules \(\phi_F\) and \(\phi_D\) on the input data in Table 1.1. With larger errors, scalable graph systems, such as Pregel, can be used to improve the ability to process large violation graphs with lower overheads. The distribution of the detected errors, however, is usually random and unpredictable; it causes the violation graph to have an irregular structure and an unequal distribution. These types of issues prevent distributed graph systems from running efficiently by creating an unbalanced workload. In the following chapters, I discuss each scalability problem in more detail.

### 1.1 Dealing with Big Datasets

The first issue in handling big datasets is that none of the existing systems can efficiently scale violation detection. This is because detecting errors, e.g., with \(\phi_F\) and
\( \phi_D \), or duplicates, \( e.g. \), with \( \phi_U \), is a combinatorial problem that quickly becomes very expensive with large sizes. More specifically, if \( |D| \) is the number of tuples in a dataset \( D \) and \( n \) is the number of tuples a given rule is defined on (\( e.g. \), \( n = 2 \) for the above rules), the time complexity of detecting errors is \( O(|D|^n) \). This high complexity leads to costly computations over large datasets, limiting the applicability of data cleansing systems. For instance, memory-based approaches \citep{5,6} report performance numbers up to 100K tuples, while a disk-based approach \citep{7} reports numbers up to 1M records only. Apart from the problem of scaling, implementing such rules in a distributed processing platform requires expertise both in the understanding of the quality rules and in the tuning of their implementation over this platform. Some research efforts have targeted the usability of a data cleansing system (\( e.g. \), NADEEF \citep{5}), but at the expense of performance and scalability where it is not practical to implement a scalable code for every possible declarative and procedural rule. Similarly, it is challenging to parallelize repair algorithms. This is because repair algorithms are complex; their primary goal is to achieve optimal and deterministic repairs at the cost of high data dependency and inseparable computations.

Therefore, designing a distributed data cleansing system to deal with big data faces two main challenges:

(1) **Scalability.** Quality rules are varied and complex: they might compare multiple tuples, use similarity comparisons, or contain inequality comparisons. Detecting errors in a distributed setting may lead to shuffling large amounts of data through the network, with negative effects on the performance. Moreover, existing repair algorithms were not designed for distributed settings. For example, they often represent the detected errors as a graph, where the graph nodes are data values and the graph edges indicate data errors among the nodes. Devising algorithms on such a graph in distributed settings is not well addressed in the literature.

(2) **Abstraction.** In addition to supporting traditional quality rules (\( e.g. \), \( \phi_F \) and \( \phi_D \)),
users also need the flexibility to define rules using procedural programs (UDFs), such as $\phi_U$. However, effective parallelization is hard to achieve with UDFs, since they are handled as black boxes. To enable scalability, finer granular constructs for specifying the rules are needed. An abstraction for this specification is challenging as it would normally come at the expense of generality of rule specification.

1.2 Sophisticated Cleaning Jobs

Consider an example of a cleaning job that requires analyzing revenue and utilization trends from different regions in an international provider of cloud services. In particular, the job requires finding out all those transactions from the West-Coast that last longer and produce smaller revenues than any transaction in the East-Coast. This can be considered as a business rule that assumes a violation if a customer from the West-Coast rented a virtual machine for more hours than customers from the East-Coast and paid less. Figure 1.2 illustrates a data instance for both tables. Assuming the customer data is stored in a database, the following query finds the violations of the business rule:

$$Q_t: \text{SELECT east.id, west.t_id FROM east, west}$$
$$\text{WHERE east.dur < west.time AND east.rev > west.cost;}$$

Although join is by far the most important and most studied operator in relational algebra [12], running $Q_t$ over 200K transactions in PostgreSQL requires more than two hours to compute the result. When analyzing the execution plan for $Q_t$ in PostgreSQL, it turns out that $Q_t$ was processed as a Cartesian product followed by a selection predicate. Since Cartesian product is the most expensive algebra operation [12], the evaluation of $Q_t$ becomes problematic due to the huge number of unnecessary intermediate results.

Even running $Q_t$ on a distributed database will not solve the complexity problem of such join queries. Moreover, despite the prevalence of these kind of queries in
Figure 1.2: East-Coast and West-Coast transactions

applications, such as temporal and spatial databases, no off-the-shelf efficient solutions exist. There have been countless techniques to optimize the different flavors of joins in various settings [13]. In the general case of a theta join, these techniques are mostly based on two assumptions: (i) one of the relations is small enough to fit in memory, and (ii) queries contain selection predicates or an equijoin with a high selectivity, which would reduce the size of the relations to be fed to the inequality join. The first assumption does not hold when joining two big relations. The second assumption is not necessarily true with low-selectivity predicates, such as gender or region, where the obtained relations are still very large. Furthermore, similar to $Q_t$ and $\phi_D$, there is a large spectrum of applications where the above two assumptions do not necessarily hold. For example, for data analytics in a temporal database, a typical query would be to find all employees and managers that overlapped while working in a certain company [14].

Common ways of optimizing such queries include sort-merge joins [15] and interval-based indexing [16–18]. Sort merge join reduces the search space by sorting the data based on the joining attributes and merging them. However, it still has a quadratic complexity for queries with inequality joins only. Interval-based indexing reduces the search space of such queries even further by using bitmap interval indexing [17]. However, such indices require large memory space [19] and long index building time. Moreover, a user would have to create multiple indices to cover all those attributes referenced in a query workload. Such indices can be built at query time, but their long construction time renders them impractical.
As a result, it is important to develop a new algorithm to efficiently handle inequality joins for $\phi_D$ and $Q_t$ for both small and big datasets. The main challenge is to be able to avoid the quadratic complexity of the join while preserving the correctness of the algorithm. Moreover, the algorithm should be general and easy to implement in both databases and data cleaning systems.

1.3 Managing Large Random Violation Graphs

With the increasing emphasis on big data, new platforms are being proposed to better exploit the structure of both the data and algorithms (e.g., Pregel [20], HADI [21], PEGASUS [22] and X-RIME [23]). Recently, Pregel [20] was introduced as a message passing-based programming model specifically targeting the mining of large graphs. For a wide array of popular graph mining algorithms (including PageRank, shortest paths problems, bipartite matching, and semi-clustering), Pregel was shown to improve the overall performance by 1-2 orders of magnitude [24] over a traditional MapReduce [25] implementation. Pregel builds on the Bulk Synchronous Parallel (BSP) [26] programming model. It operates on graph data, consisting of vertices and edges. Each vertex runs an algorithm and can send messages—asynchronously—to any other vertex. The computation is divided into several supersteps (iterations), each separated by a global synchronization barrier. During each superstep, vertices run in parallel across a distributed infrastructure. Each vertex processes incoming messages from the previous superstep. This process continues until all vertices have no messages to send, thereby becoming inactive.

Graphs have been used in data cleansing research [6, 9, 27, 28] to represent the complex relationship of detected errors. They help to combine the heterogeneous results of different data cleansing rules into a single instance, to enable a holistic view of the errors and to help users achieve higher quality repairs. The nodes of this graph represent data entities (i.e., tuples), while an edge between two nodes represents a
specific inconsistency between two data entities. The direction of the edge explains the nature of the violation. For example, as shown in Figure 1.1, the directed edge between \( t_1 \) and \( t_2 \) means that \( t_1 \) is in violation with \( t_2 \) for rule \( \phi_D \) but the opposite is not true. On the other hand, the undirected edge between \( t_2 \) and \( t_4 \) means that the violation order does not matter, where \( t_2 \) is in violation with \( t_4 \) for \( \phi_F \) and \( t_4 \) is also in violation with \( t_2 \) for the same rule. Since the number of errors grows proportionally with the size of the dirty data, it is natural to use Pregel to pre-process large violation graphs before passing them to the user’s repair algorithm. However, violation graphs are random, the structure of these graphs cannot be predicted a priori where a skewed distribution of the graph may cause computation imbalance for Pregel.

Not surprisingly, balanced computation and communication is fundamental to the efficiency of a Pregel system. To this end, existing implementations of Pregel (including Giraph [29], GoldenOrb [30], Hama [31], Surfer [32]) primarily focus on efficient partitioning of input data as a pre-processing step. They take one or more of the following five approaches to achieving a balanced workload: (1) provide simple graph partitioning schemes, like hash- or range-based partitioning (e.g., Giraph), (2) allow developers to set their own partitioning scheme or pre-partition the graph data (e.g., Pregel), (3) provide more sophisticated partitioning techniques (e.g., GraphLab, GoldenOrb and Surfer use min-cuts [33]), (4) utilize distributed data stores and graph indexing on vertices and edges (e.g., GoldenOrb and Hama), and (5) perform coarse-grained load balancing (e.g., Pregel).

Existing workload balancing approaches suffer from poor adaptation to variability (or shifts) in computation needs. Even the dynamic partition assignment that was proposed in Pregel reacts poorly to highly dynamic algorithms. I am, thus, interested in building a system that is (1) adaptive, (2) agnostic to the graph structure, and (3) requires no a priori knowledge of the behavior of the algorithm. At the same time, I am interested in improving the overall performance of the system.
1.4 Contributions and dissertation Organization

The dissertation is organized as following: In Chapter 2, I describe the related work for data cleaning, inequality joins and general-purpose scalable graph processing. After that, the specific contributions of my work are introduced as follows:

Chapter 3 presents BigDansing, a scalable data cleaning system. It abstracts the violation detection process as a logical plan composed of a set of logical operators. This abstraction allows users to detect errors and find possible fixes w.r.t. a large variety of rules that cannot be expressed in declarative languages. Once finalized, BigDansing optimizes the logical plan for efficient execution and transforms it depending on the underlying general-purpose distributed system. As a result, BigDansing allows users to focus on the logic of their rules, declaratively and procedurally, rather than on the details of efficiently parallelizing them. The abstraction enables several optimizations. I present techniques to translate a logical plan into an optimized physical plan, including: (i) removing data redundancy in input rules and reducing the number of operator calls; (ii) specialized operators to speed up the cleansing process; (iii) a new join algorithm, based on partitioning and sorting data, as an initial step, to efficiently perform distributed inequality self-joins. Since the violation detection phase is parallelized, the scalability bottleneck of big data cleansing shifts to centralized violation repairs. Therefore, I show how BigDansing is able to reduce the overhead of centralized violation repairs by utilizing violation graphs to run them in parallel without modifying the repair algorithm.

Chapter 4 introduces IEJoin, a new algorithm that utilizes bit-arrays and positional permutation arrays to achieve fast inequality joins. Given the inherent quadratic complexity of inequality joins, IEJoin follows the RAM locality is King principle coined by Jim Gray. Its use of memory-contiguous data structures, with small footprint, leads to orders of magnitude performance improvement over the prior art. The basic idea
of my proposal is to create a sorted array of tuples for each inequality comparison and compute their intersection. The prohibitive cost of the intersection operation is alleviated using a permutation array to encode positions of tuples in one sorted array w.r.t. the other sorted arrays. A bit-array is then used to emit the join results. I extend my study for IEJoin by proposing a selectivity estimation approach to efficiently process multiple predicate queries and multi-way inequality joins. I also present an incremental IEJoin to handle streaming and continuous inequality joins.

Chapter 5 introduces MIZAN, a Pregel-based graph system that uses dynamic vertex migration based on runtime monitoring of vertices to optimize the end-to-end computation. It monitors runtime characteristics of all vertices (i.e., their execution time, and incoming and outgoing messages). Using these measurements at the end of every superstep, the system constructs a migration plan that minimizes the variations across workers by identifying which vertices to migrate and where to migrate them to. This plan is simple; it does not promise optimal graph partitioning or any convergence guarantees to avoid overloading the synchronization barrier with costly computations. MIZAN, finally, uses the simple migration plan to perform a low-cost and efficient vertex migration across workers, leveraging a distributed hash table (DHT)-based location service to track the movement of vertices as they migrate. MIZAN is adaptive, agnostic to the graph structure which allows it to be efficient when processing random skewed graphs, and does not require a priori knowledge of the algorithm’s behavior which makes it suitable for processing violation graphs and any other applications.

The contributions of my dissertation are published in highly ranked venues. More specifically, the contribution of Chapter 3 (BigDansing) is published in the Proceedings of the ACM International Conference on Management of Data (SIGMOD 2015) [34]. The work described in Chapter 4 (IEJoin) is published in the Proceedings of the International Conference on Very Large Data Bases (VLDB 2015) [35] and in
the Very Large Data Bases Journal (VLDBJ 2017) [36]. Finally, my contribution in Chapter 5 (MIZAN) is published in the Proceedings of the ACM European Conference on Computer Systems (EuroSys 2013) [37].
Chapter 2

Related Work

In this chapter, I am presenting a handful of relevant research related to my dissertation. Chapter 2.1 discusses the data cleansing problem while Chapter 2.2 analyses existing techniques to efficiently scale inequality joins. Chapter 2.3, on the other hand, describes general-purpose graph processing systems and their ability to provide a balanced workload on a random graph.

2.1 Data Cleaning

Data cleansing, also called data cleaning or scrubbing, has been a topic of research in both academia and industry for decades, e.g., [4–10, 38–44]. Given some “dirty” dataset, the goal is to find the most likely errors and a possible way to repair these errors is to obtain a “clean” dataset. Although data cleansing is critical, end-to-end automatic error detection and repair systems are rarely studied in the literature.

Error detection. The work in industry, e.g., IBM QualityStage, SAP BusinessObjects, Oracle Enterprise Data Quality, and OpenRefine (previously Google Refine) has mainly focused on the use of low-level ETL (Extract, Transform and Load) rules [45]. These systems do not support UDF-based quality rules in a scalable fashion. Closer to my work, on the other hand, is NADEEF [5]. It is a generalized data cleansing system that detects violations of various data quality rules, e.g., FD, DC and UDFs, in a unified programming interface. However, NADEEF is a single machine solution; it does not scale violation detection and repairs for very large datasets. Similar to
NADEEF, SampleClean [46] is another system that aims at improving the accuracy of aggregate queries by performing cleansing over small samples of the source data. SampleClean focuses on obtaining unbiased query results with confidence intervals, while my dissertation focuses on providing a scalable framework for data cleansing. In fact, one cannot use SampleClean in traditional query processing where the entire input dataset is required.

The query optimization of declarative languages, i.e., FD, CFD and DC, and their transformation into executable codes have been heavily studied in the literature [12]. On the logical level, algebraic equivalences are used to transform first-order logic into a much more efficient variation [47–49]. On the other hand, physical access and buffer management techniques are utilized to optimize queries on the physical level [50]. These optimizations are thoroughly discussed in research assuming a relational database on the limited memory single machine. Unlike the work in this dissertation, they do not consider emerging technologies based on parallel or distributed environments.

One way to provide a scalable violation detection and repair is to use scalable data processing platforms, such as MapReduce [51] or Spark [52]. However, coding the violation detection process on top of these platforms is a tedious task that requires technical expertise. Dedoop [53], for example, is a system that detects duplicates in relational data by using Hadoop. It exploits data blocking and parallel execution to improve its performance, where it maintains its own data partitioning and distribution across workers. Although Dedoop works well for detecting duplicates, it neither provides support for scalable validation of UDFs, nor scales repair algorithms. One could also use a declarative system (e.g., Hive [54], Pig [55], or Shark [56]) on top of one of these platforms and re-implement the data quality rules using its query language. However, many common rules, such as rule \( \phi_U \), go beyond declarative formalisms. Moreover, these frameworks do not natively support inequality joins.
Theta-joins, which have been largely studied in the database community \cite{15,57}, can be used in scalable data processing platforms to support inequality joins. Studies vary from low-level techniques, such as minimizing disk accesses by choosing partitioning elements through sampling \cite{15}, to how to map arbitrary join conditions to Map and Reduce functions \cite{57}. These proposals are orthogonal to the proposed approach to handle inequality join in BigDANSING (OCJoin in Chapter 3.3.3). In fact, in my system, they rely on the executor level: if they are available in the underlying data processing platform, they can be exploited when translating the OCJoin operator at the physical level.

**Error repair.** Recent work in research for data repair include cleaning algorithms for DCs \cite{6,7} and other fragments, such as FDs and CFDs \cite{4,5,7,58,59}. These proposals also focus on specific logical constraints in a centralized setting, without much regard to scalability and flexibility. Another class of repair algorithms use machine learning tools to clean the data. Examples include SCARE \cite{60} and ERACER \cite{61}. There are also several efforts to include users (experts or crowd) in the cleaning process \cite{62,63}. Both lines of research are orthogonal to the work of my dissertation. In database management systems, *integrity constraints* and *dependencies* (e.g., inclusion dependencies and functional dependencies) are developed to add a semantic property to the relational model \cite{12}. They are supposed to be satisfied in all instances of the database schema where violating transactions are automatically rejected to maintain the database consistency.

The repair process is an unchallenging process when dealing with a single detection rule. However, when dealing with multiple rules that are logically implicated or transitive, *i.e.*, share attributes, it becomes a challenging problem since fixing an error for one dependency might introduce newer errors for other dependencies. It is the job of the repair algorithm to determine the logical implication among detection rules to apply smarter repairs. In NADEEF \cite{5}, errors are only fixed once; they
are marked as immutable after the first fix to prevent logically implicated rules from continuously generating newer errors. This approach is simple but does not guarantee repair optimality. In database systems, Chase procedure [64,65] is used to determine the logical implication in dependencies, optimize conjunctive queries and to reason about the consistency of a data design. This procedure has been picked up by data cleaning research [2,7,66–68] to compute minimal repairs in FD, CFD and Matching dependencies. The work of Kolahi et al. [9] and Chu et al. [6] also use a similar approach to minimize the number of repairs in FD and DC. They first generate a conflict hypergraph to model the association between the violations, and then utilize minimum vertex cover on the generated hypergraph to pinpoint errors with the highest impact. By only fixing those errors, they succeed in reducing the number of required changes to the input dataset.

Representing detected violations as a graph has already been addressed in research. Other than the work of Kolahi et al. [9] and Chu et al. [6], Ebaid et al. [27] and Elmagarmid [28] used violation graphs in NADEEF to visualize and allow user interaction with violations related to FDs, CFDs, matching dependency (MD) and entity resolution. None of those systems utilized the properties of the violation graph, as discussed in Chapter 3.4, to scale data repairing algorithms in distributed systems.

2.2 Scaling Inequality Joins

Several cases of inequality joins have been studied in the literature; these include band joins, interval joins and, more generally, spatial joins. IEJoin is specially optimized for joins with at least two predicates in \{“<”, “>”, “≤”, “≥”, “≠”\}.

A band join [15] of two relations \(R\) and \(S\) has a join predicate that requires the join attribute of \(S\) to be within some range of the join attribute of \(R\). The join condition is expressed as \(R.A - c_1 \leq S.B \& S.B \leq R.A + c_2\), where \(c_1\) and \(c_2\) are constants. The band-join algorithm [15] partitions the data from relations \(R\) and \(S\)
into partitions $R_i$ and $S_i$ respectively, such that for every tuple $r \in R$, all tuples of $S$ that join with $r$ appear in $S_i$. It assumes that $R_i$ fits into memory. Contrary to IEJoin, the band join is limited to a single inequality condition type, involving one single attribute from each column. IEJoin works for any inequality conditions and attributes from the two relations. While band join queries can be processed using my algorithm, not all IEJoin queries can run with a band join algorithm.

Interval joins are frequently used in temporal and spatial data. The work in [16] proposes the use of the relational Interval Tree to optimize joining interval data. Each interval intersection is represented by two inequality conditions, where the lower and upper times of any two tuples are compared to check for overlaps. This work optimizes non-equijoins on interval intersections, where they represent each interval as a multi-value attribute. Compared to my work, they only focus on improving interval intersection queries and cannot process general-purpose inequality joins.

Spatial indexing is widely used in several applications with multidimensional datasets, such as Bitmap indices [69, 70], R-trees [71] and space filling curves [72]. In PostgreSQL, support for spatial indexing algorithms is provided through a single interface known as Generalized index Search Tree [18] (GiST). From this collection of indices, Bitmap index is the most suitable technique to optimize multiple attribute queries that can be represented as 2-dimensional data. Examples of 2-dimensional datasets are intervals (e.g., start and end time in $Q_2$), GPS coordinates (e.g., $Q_3$), and any two numerical attributes that represent a point in an XY plot (e.g., salary and tax in $Q_1$). The main disadvantage of the Bitmap index is that it requires a large memory footprint to store all unique values of the composite attributes [17, 19]. Bitmap index is a natural baseline for my algorithm, but, unlike IEJoin, it does not perform well with high cardinality attributes, as demonstrated in Figure 4.9. R-trees, on the other hand, are not suitable because an inequality join corresponds to window queries that are unbounded from two sides, and consequently intersect with a large
number of internal nodes of the R-tree, generating unnecessary disk accesses.

The patent in [73] also presents an algorithm to optimize the Cartesian product when joining two tables based on a single inequality condition. The algorithm partitions the input relations into smaller blocks based on the value distribution and min and max values of the join predicate. The algorithm then applies a Cartesian product on a subset of the input partitions, where it eliminates unnecessary partitions depending on the join condition. Compared with my approach, this algorithm optimizes the Cartesian product through partitioning based on only one single inequality join predicate.

Several other proposals have been made to speedup join executions in MapReduce (e.g., [74]). However, they focus on joins with equalities thus requiring massive data shuffling to be able to compare each tuple with each other. There have been few attempts to devise efficient implementation of theta-join in MapReduce [57, 75]. [57] focuses on pair-wise theta-join queries. It partitions the Cartesian product output space with rectangular regions of bounded sizes. Each partition is mapped to one reducer. The proposed partitioning guarantees correctness and workload balance among the reducers while minimizing the overall response time. [75] further extends [57] to solve multi-way theta-joins. It proposes an I/O and network cost-aware model for MapReduce jobs to estimate the minimum time execution costs for all possible decomposition plans for a given query, and selects the best plan given a limited number of computing units and a pool of possible jobs. I propose a new algorithm to do the actual inequality join based on sorting, permutation arrays, and bit arrays. The focus in these previous proposals is on efficiently partitioning the output space and on providing a cost model for selecting the best combination of MapReduce jobs to minimize response time. In both proposals, the join is performed with existing algorithms, where inequality conditions correspond to a Cartesian product followed by a selection.
Fagin’s Algorithm has a similar data access pattern to the bit-array scanning of IEJoin (Chapter 4.1). Both algorithms sort the database objects’ (or Tuples’) attributes into multiple lists, where each list is sorted based on a distinct attribute. After that, the content of an object’s attribute is extracted from the attribute-sorted lists by randomly accessing the lists based on the object’s key. Although Fagin’s Algorithm and IEJoin share the same data access pattern, they have different objectives. Fagin’s Algorithm extracts objects from the sorted attribute lists. Each extracted object is assigned a score by applying a monotonic aggregation function on its attributes. The algorithm then returns a sorted list, based on the objects’ score, that represents the top-K objects in the database for some query. IEJoin, on the other hand, results in a list of object pairs that satisfies some inequality condition between two sets of attributes. It utilizes a bit-array on top of the two attribute-sorted lists to minimize the cost of pair-wise comparisons.

Regarding selectivity estimation, there are lots of approaches focusing on this problem. However, existing work on selectivity estimation are mostly focused on equijoins. There have been few proposals for the general case of theta-join and spatial join. Nevertheless, most of these proposals estimate the selectivity of the inequality join to be $O(n^2)$ since it is evaluated as a Cartesian product. IEJoin significantly differs from the related work as it is not considered a Cartesian product. It uses a simple, yet highly efficient, selectivity estimation technique that computes the number of overlapping sorted blocks obtained from a sample of input relations.

### 2.3 General-purpose Scalable Graph Processing

Graph processing is a challenging problem; Lumsdaine et al. show that the graph’s high data access to computation ratio makes the performance and scalability of parallel graph processing systems vulnerable to graph irregularity and poor data locality. In the past few years, a number of large scale graph processing systems have been pro-
posed to address these challenges. A common thread across the majority of existing work is developing high-level expressive graph processing abstractions, and parallelize them while researching various types of static graph partitioning to balance computation across a distributed workers. However, almost none of the graph processing systems touched on the problem of applying dynamic load balancing through runtime graph partitioning, which Bruce et al. [84] assert to be either too expensive or hard to parallelize.

**MapReduce-based graph processing systems.** MapReduce [25] is a programming abstraction proposed by Google to simplify efficient large data processing. It provides automatic parallelization, failure recovery and load balancing to users who don’t have any parallel programming experience. Hadoop [85] –the open-source version of MapReduce– has been used by both PEGASUS [22] and X-Rime [23, 86] to provide a large scale graph processing system, through a fixed set of highly optimized graph algorithm implementations on top of the MapReduce programming model. Since a handful of graph algorithms can be represented as a matrix-vector computation, PEGASUS implemented a generalization of matrix-vector multiplication on top of Hadoop to facilitate graph processing. On the other hand, X-Rime implemented an optimized graph data store on top of HDFS, and utilized the scalability of Hadoop to distribute the graph data store among a large number of Hadoop workers to improve large scale graph processing. The main advantage of PEGASUS and X-Rime is the utilization of existing scalable and reliable general data processing platforms to provide graph processing systems instead of the overhead of implementing them from scratch. However, the dependency on Hadoop comes with a price; both systems are affected by Hadoop’s inefficiency in iterative computations and a limitation on processing skewed data. Moreover, graph algorithms on top of PEGASUS and X-Rime require careful implementation with lots of optimization tricks, which complicates programming for general users. Improving graph processing implemented on top of
MapReduce requires improving the work flow of MapReduce itself, which is a well studied problem in literature [87]. Regardless of the possible MapReduce optimizations, its programming abstraction is too generic and does not fit the nature of graph algorithms.

**Pregel and its clones.** Pregel [20] is a message passing graph processing system introduced by Google to overcome the limitations of graph processing on top of MapReduce. Its programming abstraction is built on top of the Bulk Synchronous Parallel (BSP) [26] programming model. Kajdanowicz et al. [88] surveyed the difference between MapReduce and BSP-based graph processing systems. In Pregel, each vertex runs a user-defined algorithm independently and can send messages asynchronously to any other vertex. Computation is divided into a number of supersteps (iterations), each separated by a global synchronization barrier. During each superstep, vertices run in parallel across a distributed infrastructure. Each vertex processes incoming messages from the previous superstep. This process continues until all vertices have no messages to send, thereby becoming inactive. Pregel overlaps vertex computations and communication to provide efficient parallel execution for large distributed graphs. The default partitioning scheme for the input graph used by Pregel is hash-based, where it assigns more than one subgraph (partition) to a worker. Figure 2.1 shows an
example of hash-based partitioning with three partitions and seven edge cuts. Pregel claims that its programming model is expressive enough to represent a wide spectrum of graph algorithms while the user’s code is inherently free of deadlocks. Fault tolerance in Pregel is provided through synchronously checkpointing the graph’s data and vertices’ messages frequently. Pregel is considered a scalable abstraction for graph processing, but it lacks runtime load balancing.

Giraph \cite{29} is an open source clone for Pregel that is built on top of Hadoop. It supports Pregel’s programming model in addition to a set of other graph processing features developed by the community. Giraph supports hash-based and range-based partitioning schemes; the user can also implement his own custom graph partitioning scheme. Giraph, for the time being, inherits all the limitations of Pregel besides the overhead imposed by Java’s garbage collector.

GoldenOrb \cite{30} is another clone of Pregel that uses Hadoop’s file system (HDFS) as its storage system and employs hash-based partitioning. They claim that they are able to dynamically increase the storage and processing during runtime, but they don’t provide any technical details about this feature. Similar to Giraph, GoldenOrb also inherits the limitations of Pregel. An optimization that was introduced in \cite{89} uses a fast approximate min-cuts graph partitioning algorithm known as METIS \cite{33},

Figure 2.2: Example of three-way min-cuts partitioning using METIS with four edge cuts.
instead of GoldenOrb’s hash-based partitioning, to maintain a balanced workload among the partitions and minimize communication across workers. Figure 2.2 shows an example of applying three-way METIS partitioning on a sample graph which reduced 40% of the edge cuts compared to hash-based partitioning in Figure 2.1. Additionally, they propose to utilize a distributed data storage with replication to improve the data access speed of the frequently accessed vertices for the distributed graph processing system.

Similarly, Surfer [32] utilizes min-cut graph partitioning to improve the performance of Pregel’s computations. However, Surfer focuses more on providing a bandwidth-aware variant of the multilevel graph partitioning algorithm (integrated into a Pregel implementation) to improve the performance of graph partitioning on the cloud. These systems focus on providing static graph partitioning techniques to optimize graph processing and reduce the distributed communication cost, but lack dynamic runtime optimizations required for algorithms with non-predictable runtime behavior. Such algorithms can cause imbalance to the workers’ computation and message traffic during runtime that leads to slower algorithm execution. They also lack the support for efficient processing of highly skewed power-law graphs.

A recent work done by Shang et al. [90] optimizes Pregel’s runtime by looking into the dynamic behavior of algorithms. They constructed an optimization model called Wind, implemented on top of an open source bulk synchronous parallel computing framework known as HAMA [31], based on the fact that graph algorithms do not access all vertices in every superstep. Wind classified graph algorithms into three categories: Always-Active-Style, Traversal-Style, and Multi-Phase-Style. It also monitors the active set of vertices that requires to run the graph algorithm and apply fine-grain vertex migrations to minimize the communication while maintaining work-load balance. This work is relatively similar to my work done in Mizan [37], that is discussed in Chapter 5, but they diverge in defining the optimization objective of
vertex migrations.

**Shared memory graph processing systems.** GraphLab [91] is another programming abstraction proposed to overcome the limitations of MapReduce. It provides efficient parallel processing for graph and machine learning algorithms. Unlike BSP frameworks, GraphLab supports both asynchronous and synchronous processing where the computation is executed on the most recently available data. That is, each vertex in GraphLab runs the user code and modifies its data or the data of other vertices asynchronously through a shared memory abstraction. GraphLab ensures that its asynchronous multi-threaded computations are consistent and can provide equal results to the serial execution by providing a set of consistency models to eliminate the race conditions caused by overlapping computations. Moreover, users of GraphLab have the choice of improving the performance of their algorithms and the cost of computation consistency. Distributed GraphLab [92] extends the original multi-core GraphLab implementations to a cloud-based distributed GraphLab, where they apply a two-phase partitioning scheme that uses ParMetis [93] to minimize the edges among graph partitions, and allows for a fast graph repartitioning. Failure recovery in GraphLab is provided through asynchronous fixed-interval graph snapshots.

Similar to GraphLab, HipG [94] is designed to operate on distributed memory machines while providing transparent memory sharing to support hierarchical parallel graph algorithms. HipG also processes graph algorithms using the BSP model. Unlike the global synchronous barriers of Pregel, HipG applies more fine-grained barriers, called synchronizers, during algorithm execution. The user has to be involved in writing the synchronizers; thus, it requires additional complexity in the users’ code.

GraphChi [95] is a disk based graph processing system based on GraphLab and is designed to run on a single machine with limited memory. It requires a small number of non-sequential disk accesses to process the subgraphs from disk, while allowing for asynchronous iterative computations. These systems neither support any dynamic
load balancing techniques, nor support processing graphs with power-law.

**Power-law optimized graph processing systems.** Stanford GPS (GPS) \[96\] is another clone to Pregel but has three additional features. First, GPS extends Pregel’s API to handle graph algorithms that perform both vertex-centric and global computations. Second, GPS supports runtime dynamic graph repartitioning based on the objective to minimize communication from any given initial graph partitioning. Third, GPS provides an optimization scheme called *large adjacency list partitioning*, where any high degree vertex is replicated across workers and its out-degree list is partitioned and distributively stored by the replicas to minimize the effect of graphs with power-law distributions. The runtime dynamic graph repartitioning in GPS only monitors outgoing messages to balance the network I/O across workers. However, it does not fully capture the dynamic behavior of different algorithms that can be caused by other factors than those of the outgoing messages. Moreover, GPS lacks the flexibility of migrating vertices to other workers since it depends on vertex hashing to locate vertices in the distributed cluster, while it requires system wide vertex relabeling for migrated vertices.

PowerGraph \[97\], on the other hand, is a distributed graph system derived from GraphLab that overcomes the challenges of processing natural graphs with extreme power-law distributions. PowerGraph introduces vertex-based partitioning with replication to distribute the graph’s data based on its edges. It also provides a programming abstraction (based on gather, apply and scatter operators) to distribute the user’s algorithm workload on their graph processing system. PowerGraph focused on evaluating static algorithms on power-law graphs. It is, thus, unclear how their system performs across the broader class of algorithms with variable behaviors and graphs (e.g., non-power-law). The programming abstraction of PowerGraph that improves power-law graph processing, supports only commutative and associative graph algorithms, or graph algorithms that support message combiners, which can be more
restrictive than Pregel’s and would add extra overhead to graph algorithm developers.

**Specialized graph systems.** Kineograph [98] is a distributed system that targets the processing of fast changing graphs. It captures the relationships in the input data, reflects them on the graph structure, and creates regular snapshots of the data. Data consistency is ensured in Kineograph by separating graph processing and graph update routines. This system is mainly developed to support algorithms that mine the incremental changes in the graph structure over a series of graph snapshots. The user can also run an offline iterative analysis over a copy of the graph, similar to Pregel. The Little Engine(s) [99] is also a distributed system designed to scale with online social networks. This system utilizes graph repartitioning with one-hop replication to rebalanced any changes to the graph structure, localize graph data, and reduce the network cost associated with query processing. The Little Engine(s) repartitioning algorithm aims to find min-cuts in a graph under the condition of minimizing the replication overhead at each worker. This system works as a middle-box between an application and its database systems; it is not intended for offline analytics or batch data processing like Pregel or MapReduce. Ligra [100] is another parallel shared memory based graph processing system that provides programming abstractions, highly customized for graph traversal algorithms on large multicore machines. It requires users to provide two functions that are mapped on subsets of vertices and edges respectively and independently. Compared to Pregel and GraphLab, Ligra is a domain specific graph processing system and does not target general graph processing problems.

Trinity [101] is a distributed in-memory key-value storage based graph processing system that supports both offline analytics and online query answering. Trinity provides a similar—but more restrictive—computing model to Pregel for offline graph analytics. The key advantage of Trinity is its low latency graph storage design, where the data of each vertex is stored in a binary format to eliminate decoding and encoding
costs.

Similar to Trinity, STAPL parallel graph library (SGL) \textsuperscript{102} is an in-memory distributed graph processing system that abstracts data storage and parallelism complexity from users. The main contribution of SGL is providing universal access to the distributed in-memory graph data through a shared object to the user’s code. SGL also supports multiple graph processing abstractions including a synchronous and an asynchronous coarse paradigm such as PBGL \textsuperscript{103} and fine-grained paradigms such as Pregel and GraphLab. Unlike Trinity, SGL supports graph redistribution and vertex migrations asynchronously during runtime to rebalance the data shared object through a user-defined cost function. While these systems address specific design challenges in processing large graphs, their focus is orthogonal to my work in Chapter \textsuperscript{5}.
Chapter 3

BigDansing: A System for Big Data Cleansing

In this chapter, I present BigDansing, a Big Data Cleansing that is easy-to-use and highly scalable. It provides automatic task parallelization for cleansing jobs, which translates the user’s input rule into execution plans on top of general-purpose scalable data processing platforms. I start by defining BigDansing’s cleansing semantics and architecture (Chapter 3.1). After that, I introduce my rule specification abstraction and its transformation to logical plans (Chapter 3.2). This abstraction allows users to focus on the logic of their rules without worrying about the details of parallelism or runtime efficiency. I then describe techniques to translate each logical plan into an optimized physical plan, and then to execution plans on both Spark and MapReduce, which ensures efficient parallel execution (Chapter 3.3). To support scalable repair in BigDansing, I consider two approaches that enable existing repair algorithms to run in distributed settings (Chapter 3.4). Finally, I evaluate BigDansing on both real-world and synthetic datasets with various rules to comprehensively validate its efficiency (Chapter 3.5).

3.1 Fundamentals and An Overview

I discuss the data cleansing semantics expressed by BigDansing and then give an overview of the system.
3.1.1 Data Cleansing Semantics

In BigDansing, the input data is defined as a set of data units, where each *data unit* is the smallest unit of input datasets. Each unit can have multiple associated *elements* that are identified by model-specific functions. For example, the database tuples are the data units for the relational model and the attributes identify their elements, while triples are the data units for RDF data (see Chapter 3.6). BigDansing provides a set of parsers for producing such data units and elements from input datasets.

BigDansing adopts UDFs as the basis to define quality rules. Each rule has two fundamental abstract functions, namely Detect and GenFix. Detect takes one or multiple data units as input, and outputs a *violation*, *i.e.*, elements in the input units that together are considered as erroneous *w.r.t.* the rule:

\[
\text{Detect}(\text{data}\_\text{units}) \rightarrow \text{violation}
\]

GenFix takes a violation as input and computes alternative, possible updates to resolve this violation:

\[
\text{GenFix}(\text{violation}) \rightarrow \text{possible}\_\text{fixes}
\]

The language of the possible fixes is determined by the capabilities of the repair algorithm. With the supported algorithms, a *possible fix* in BigDansing is an expression of the form \(x \text{ op } y\), where \(x\) is an element, \(\text{op}\) is in \(\{=, \neq, <, >, \geq, \leq\}\), and \(y\) is either an element or a constant. In addition, BigDansing has new functions to enable distributed and scalable execution of the entire cleansing process. I defer the details to Chapter 3.2.

Consider Example 1 in Chapter 1, the Detect function of \(\phi_F (D(\text{zipcode} \rightarrow \text{city}))\) takes two tuples (*i.e.*, two data units) as input, and identifies a violation whenever the same *zipcode* value appears in the two tuples but with a different *city*. Thus, \(t_2(90210, LA)\) and \(t_4(90210, SF)\) are a violation. The GenFix function could
enforce either \(t_2[\text{city}]\) and \(t_4[\text{city}]\) to be the same, or at least one element between \(t_2[\text{zipcode}]\) and \(t_4[\text{zipcode}]\) to be different from 90210. Rule \(\phi_U (\forall t_1, t_2 \in D, \neg(\text{simF}(t_1.\text{name}, t_2.\text{name}) \land \text{getCounty}(t_1.\text{city}) = \text{getCounty}(t_2.\text{city})))\) is more general as it requires special processing. Detect takes two tuples as input, and outputs a violation whenever it finds similar name values and obtains the same county values from a mapping table. GenFix could propose to assign the same values to both tuples so that one of them is removed in set semantics.

By using a UDF-based approach, I can support a large variety of traditional quality rules with a parser that automatically implements the abstract functions, e.g., CFDs [4] and DCs [6], but also more procedural rules that are provided by the user. These latter rules can implement any detection and repair method expressible with procedural code, such as Java, as long as they implement the signatures of the two above functions, as demonstrated in systems such as NADEEF [5]. Note that one known limitation of UDF-based systems is that, when treating UDFs as black-boxes, it is hard to do static analysis, such as consistency and implication, for the given rules.

**BIGDANSING** targets the following *data cleansing problem*: given a dirty data \(D\) and a set of rules \(\Sigma\), compute a repair which is an updated instance \(D'\) such that there are no violations, or there are only violations with no possible fixes.

Among the many possible solutions to the cleansing problem, it is common to define a notion of minimality based on the cost of a repair. A popular cost function [59] for relational data is the following: \(\sum_{t \in D, t' \in D', A \in A_R} dis_A(D(t[A]), D(t'[A]))\), where \(t'\) is the fix for a specific tuple \(t\), and \(dis_A(D(t[A]), D(t'[A]))\) is a distance between their values for attribute \(A\) (an exact match returns 0). This models the intuition that the higher is the sum of the distances between the original values and their fixes, the more expensive is the repair. Computing such *minimum* repairs is NP-hard, even with FDs only [9,59]. Thus, data repair algorithms are mostly heuristics (see Chapter 3.4).
3.1.2 Architecture

The architecture of BIGDANSING is illustrated in Figure 3.1. BIGDANSING receives a data quality rule together with a dirty dataset from users (1) and outputs a clean dataset (7). BIGDANSING consists of two main components: the RuleEngine and the RepairAlgorithm.

The RuleEngine receives a quality rule either in a UDF-based form (a BIGDANSING job) or in a declarative form (a declarative rule). A job (i.e., a script) defines users operations as well as the sequence in which users want to run their operations (see Chapter 3.2.2). A declarative rule is written using traditional integrity constraints. In
the latter case, the RuleEngine automatically translates the declarative rule, through the RuleParser, into a job to be executed on a parallel data processing framework. The RuleParser is a static rule-based optimizer that supports well-known integrity constraints: FD, CFD and DC. However, the generic form of first-order logic may also be supported in BigDansing, which only requires modifying the RuleParser.

This job outputs the set of violations and possible fixes for each violation. The RuleEngine has three layers: the logical, physical, and execution layers. This architecture allows BigDansing to (i) support a large variety of data quality rules by abstracting the rule specification process, (ii) achieve high efficiency when cleansing datasets by performing a number of physical optimizations, and (iii) scale to big datasets by fully leveraging the scalability of existing parallel data processing frameworks. Notice that, unlike a DBMS, the RuleEngine also has an execution abstraction, which allows BigDansing to run on top of general-purpose data processing frameworks ranging from MapReduce-like systems to databases.

(1) Logical layer. A major goal of BigDansing is to allow users to express a variety of data quality rules in a simple way. This means that users should only care about the logic of their quality rules, without worrying about how to make the code distributed. To this end, BigDansing provides five logical operators, namely Scope, Block, Iterate, Detect, and GenFix, to express a data quality rule: Scope defines the relevant data for the rule; Block defines the group of data units among which a violation may occur; Iterate enumerates the candidate violations; Detect determines whether a candidate violation is indeed a violation; and GenFix generates a set of possible fixes for each violation. Users define these logical operators, as well as the sequence in which BigDansing has to run them, in their jobs. Alternatively, users provide a declarative rule and BigDansing translates it into a job having these five logical operators. Notice that a job represents the logical plan of a given input quality rule.

(2) Physical layer. In this layer, BigDansing receives a logical plan and transforms
it into an optimized physical plan of physical operators. Like in DBMSs, a physical plan specifies how a logical plan is implemented. For example, **Block** could be implemented by either hash-based or range-based methods. A physical operator in **BIGDANSING** also contains extra information, such as the input dataset and the degree of parallelism. Overall, **BIGDANSING** processes a logical plan through two main optimization steps, namely the *plan consolidation* and the *data access optimization*, through the use of specialized join and data access operators.

**Execution layer.** In this layer, **BIGDANSING** determines how a physical plan will be actually executed on the underlying parallel data processing framework. It transforms a physical plan into an execution plan which consists of a set of system-dependent operations, e.g., a Spark or MapReduce job. **BIGDANSING** runs the generated execution plan on the underlying system. Then, it collects all the violations and possible fixes produced by this execution. As a result, users get the benefits of parallel data processing frameworks by just providing few lines of code for the logical operators.

Once **BIGDANSING** has collected the set of violations and possible fixes, it proceeds to repair (*cleanse*) the input dirty dataset. At this point, the repair process is independent from the number of rules and their semantics, as the repair algorithm considers only the set of violations and their possible fixes. The way the final fixes are chosen among all the possible ones strongly depends on the repair algorithm itself. Instead of proposing a new repair algorithm, I present in Chapter 3.4 two different approaches to implement existing algorithms in a distributed setting. Correctness and termination properties of the original algorithms are preserved in my extensions. Alternatively, expert users can plug in their own repair algorithms. In the algorithms that I extend, each repair step greedily eliminates violations with possible fixes while minimizing the cost function in Chapter 3.1.1 An iterative process, *i.e.*, detection and repair, terminates if there are no more violations or there are only violations...
with no corresponding possible fixes. The repair step may introduce new violations on previously fixed data units and a new step may decide to again update these units. To ensure termination, the algorithm places a special variable on such units after a fixed number of iterations (which is a user-defined parameter), thus eliminating the possibility of future violations on the same data.

3.2 Rule Specification

BIGDANSING abstraction consists of five logical operators: Scope, Block, Iterate, Detect, and GenFix, which are powerful enough to express a large spectrum of cleansing tasks. While Detect and GenFix are the two general operators that model the data cleansing process, Scope, Block, and Iterate enable the efficient and scalable execution of that process. Generally speaking, Scope reduces the amount of data that has to be treated, Block reduces the search space for the candidate violation generation, and Iterate efficiently traverses the reduced search space to generate all candidate violations. It is worth noting that these five operators do not model the repair process itself (i.e., a repair algorithm). The system translates these operators along with a, generated or user-provided, BIGDANSING job into a logical plan.

3.2.1 Logical Operators

As mentioned earlier, BIGDANSING defines the input data through data units $U_s$ on which the logical operators operate. While such a fine-granular model might seem to incur high cost as BIGDANSING calls an operator for each $U$, it in fact allows application of an operator in a highly parallel fashion.

In the following, I define the five operators provided by BIGDANSING and illustrate them in Figure 3.2 using the dataset and rule FD $\phi_F$ (zipcode $\rightarrow$ city) of Example 1. Notice that the following listings are automatically generated by the system for declarative rules. Users can either modify this code or provide their own
for the UDFs case.

(1) **Scope** removes irrelevant data units from a dataset. For each data unit \( U \), **Scope** outputs a set of filtered data units, which can be an empty set.

\[
\text{Scope}(U) \rightarrow \text{list}(U')
\]

Notice that **Scope** outputs a list of \( U \)s as it allows data units to be replicated. This operator is important as it allows **BigDansing** to focus only on data units that are relevant to a given rule. For instance, in Figure 3.2, **Scope** projects on attributes zipcode and city. Listing 3.1 shows the lines of code for **Scope** in rule \( \phi_F \).

```java
public Tuple scope(Tuple in) {
    Tuple t = new Tuple(in.getID());
    t.addCell(in.getCellValue(1)); // zipcode
    t.addCell(in.getCellValue(2)); // city
    return t;
}
```

Listing 3.1: Code example for the **Scope** operator.

(2) **Block** groups data units sharing the same blocking key. For each data unit \( U \), **Block** outputs a blocking key.

\[
\text{Block}(U) \rightarrow \text{key}
\]

The **Block** operator is crucial for **BigDansing**’s scalability as it narrows down the number of data units on which a violation might occur. For example, in Figure 3.2, **Block** groups tuples on attribute zipcode, resulting in three blocks, with each block having a distinct zipcode. Violations might occur inside these blocks only and not across blocks. Listing 3.2 shows the single line of code required by this operator for rule \( \phi_F \).

(3) **Iterate** defines how to combine data units \( U \)s to generate candidate violations.
public String block(Tuple t) {
    return t.getCellValue(0); // zipcode
}

Listing 3.2: Code example for the Block operator.

This operator takes as input a list of lists of data units $U$s (because it might take the output of several previous operators) and outputs a single $U$, a pair of $U$s, or a list of $U$s.

\[
\text{Iterate}(\text{list(list}(U))) \rightarrow U' \mid \{U_i, U_j\} \mid \text{list}(U'')
\]

This operator allows BiGDANSING to avoid the quadratic complexity for generating candidate violations. For instance, in Figure 3.2 Iterate passes each unique combination of two tuples inside each block, producing four pairs only (instead of 13 pairs): $(t_3, t_5)$ from $B_1$, $(t_2, t_4)$, $(t_2, t_6)$, and $(t_4, t_6)$ from $B_3$. Listing 3.3 shows the code required by this Iterate operator for rule $\phi_F$.

public Iterable<TuplePair> Iterate(ListTupleList<String> in) {
    ArrayList<TuplePair> tp = new ArrayList<TuplePair>();
    List<Tuple> inList = in.getIterator().next().getValue();
    for (int i = 0; i < inList.size(); i++) {
        for (int j = i + 1; j < inList.size(); j++) {
            tp.add(new TuplePair(inList.get(i), inList.get(j)));
        }
    }
    return tp;
}

Listing 3.3: Code example for the Iterate operator.

(4) Detect takes a single $U$, a pair-$U$, or a list of $U$s, as input and outputs a list of violations, possibly empty. The single $U$ indicates that the Detect function should check for violations within the same data unit. The pair-$U$ is a special case of list of $U$s, however, both input types require Detect to check for violations that are collaboratively caused by the input data units. For example, check if the input list
of $U$s are all duplicates.

$$\text{Detect}(U \mid \langle U_i, U_j \rangle \mid \text{list}(U')) \rightarrow \{\text{list(violation)}\}$$

Considering three types of inputs for Detect allows me to achieve better parallelization by distinguishing between different granularities of the input. For example, having 1K $U$s as input, rather than a single list of $U$s, would allow me to run 1K parallel instances of Detect (instead of a single Detect instance). In Figure 3.2 Detect outputs two violations, $v_1 = (t_2, t_4)$ and $v_2 = (t_6, t_4)$, as they have different values for city; and it requires the lines of code in Listing 3.4.

```
public ArrayList<Violation> detect(TuplePair in) {
    ArrayList<Violation> lst = new ArrayList<Violation>();
    if (!in.getLeft().getCellValue(1).equals(in.getRight().getCellValue(1))) {
        Violation v = new Violation("zipcode => City");
        v.addTuple(in.getLeft());
        v.addTuple(in.getRight());
        lst.add(v);
    }
    return lst;
}
```

Listing 3.4: Code example for the Detect operator.
(5) **GenFix** computes a set of possible fixes for a given violation.

\[ \text{GenFix}(\text{violation}) \rightarrow \{\text{list(PossibleFixes)}\} \]

For instance, assuming that only right-hand side values can be modified, **GenFix** produces one possible repair for each detected violation (Figure 3.2): \( t_2[\text{city}] = t_4[\text{city}] \) and \( t_6[\text{city}] = t_4[\text{city}] \). Listing 3.5 shows the code for this **GenFix**.

```java
public ArrayList<Fix> GenFix(Violation v) {
    ArrayList<Fix> result = new ArrayList<Fix>();
    Tuple t1 = v.getLeft();
    Tuple t2 = v.getRight();
    Cell c1 = new Cell(t1.getID(),"City",t1.getCellValue(1));
    Cell c2 = new Cell(t2.getID(),"City",t2.getCellValue(1));
    result.add(new Fix(c1,"=",c2));
    return result ;
}
```

Listing 3.5: Code example for the **GenFix** operator.

Additionally, to better specify the data flow among the different operators, I introduce a *label* to stamp a data item and track how it is being processed.

In contrast to DBMS operators, *BIGDANSING*’s operators are UDFs, which allow users to plug in any logic. As a result, I am not restricted to a specific data model. However, for ease of explanation, all of my examples assume relational data. I report an RDF data cleansing example in Chapter 3.6. I also report in Appendix B the code required to write the same rule in a distributed environment, such as Spark. The proposed templates for the operators allow users to obtain distributed detection without any expertise on Spark, only by providing from 3 to 16 lines of Java code. The benefit of the abstraction should be apparent at this point: (i) ease-of-use for non-expert users and (ii) better scalability thanks to its abstraction.
3.2.2 BigDansing Example Job

I explain with a BigDansing job example how users can specify to BigDansing in which sequence to run their logical operators. Users can fully control the execution flow of their logical operators via data labels, which in other words represent the data flow of a specific input dataset. For example, users would write the BigDansing job in Listing 3.6 to generate the logical plan in Figure 3.4. First of all, users create a job instance of BigDansing to specify their requirements (Line 1). Users specify the input datasets they want to process by optionally providing their schema (Line 2) and their path (D1 and D2 in Lines 3 & 4). Additionally, users label input datasets to define the number data flows they desire to have (S and T for D1 and W for D2). Notice that, in this example, the job creates a copy of D1. Then, users specify the sequence of logical operations they want to perform to each data flow (Lines 5-11). BigDansing respects the order in which users specify their logical operators, e.g., BigDansing will first perform Scope and then Block1. Users can also specify an arbitrary number of inputs and outputs. For instance, Iterate1 gets data flows S and T as input and outputs a signal data flow M. As a last step, users run their job as in Line 12.

3.2.3 The Planning Process

The logical layer of BigDansing takes as input a set of labeled logical operators together with a job and outputs a logical plan. The system starts by validating whether the provided job is correct by checking that all referenced logical operators are defined and at least one Detect is specified. Recall that for declarative rules, such as CFDs and DCs, users do not need to provide any job, as BigDansing automatically generates a job along with the logical operators.

After validating the provided job, BigDansing generates a logical plan (Fig-
public void main (String args []) {
1 BigDansing job = new BigDansing("Example Job");
2 String schema = "name,zipcode,city,state,salary,rate";
3 job .addInputPath(schema,"D1", "S", "T");
4 job .addInputPath(schema,"D2", "W");
5 job .addScope(Scope, "S");
6 job .addBlock(Block1, "S");
7 job .addBlock(Block2, "T");
8 job .addIIterate ("M", Iterate1, "S", "T");
9 job .addIIterate ("V", Iterate2, "W", "M");
10 job .addDetect(Detect, "V");
11 job .addGenFix(GenFix, "V");
12 job .run();
}

Listing 3.6: Example of a user’s BigDansing job.

Figure 3.3: Planner execution flow

ure[3.3] that: (i) must have at least one input dataset \( D_i \); (ii) may have one or more 
Scope operators; (iii) may have one Block operator or more linked to an Iterate operator; (iv) must have at least one Detect operator to possibly produce a set of violations; and (v) may have one GenFix operator for each Detect operator to generate possible fixes for each violation.

BigDansing looks for a corresponding Iterate operator for each Detect. If Iterate is not specified, BigDansing generates one according to the input required by the Detect operator. Then, it looks for Block and Scope operators that match the input label of the Detect operator. If an Iterate operator is specified, BigDansing identifies all Block operators whose input label match the input label of the Iterate. Then, it uses the input labels of the Iterate operator to find other possible Iterate operators in reverse order. Each new detected Iterate operator is added to the plan with the Block
operators related to its input labels. Once it has processed all Iterate operators, it finally looks for Scope operators.

In case the Scope or Block operators are missing, BigDansing pushes the input dataset to the next operator in the logical plan. If no GenFix operator is provided, the output of the Detect operator is written to disk. For example, let’s consider the logical plan in Figure 3.4 which is generated by BigDansing when a user provides the job in Chapter 3.2.2. I observe that dataset D1 is sent directly to a Scope and a Block operator. Similarly, dataset D2 is sent directly to an Iterate operator. I also observe that one can also iterate over the output of previous Iterate operators (e.g., over output D_M). This flexibility allows users to express complex data quality rules, such as in the form of “bushy” plans as shown in Appendix A.

3.3 Building Physical Plans

When translating a logical plan, BigDansing exploits two main opportunities to derive an optimized physical plan: (i) static analysis of the logical plan and (ii) alternative translations for each logical operator. Since quality rules may involve joins over ordering comparisons, I also introduce an efficient algorithm for these cases. I consider these aspects in this chapter. In addition, I discuss the translation of physical plans to Spark and MapReduce while describing the data storage and access issues of BigDansing.
3.3.1 Physical Operators

There are two kinds of physical operators: *wrappers* and *enhancers*. A wrapper simply invokes a logical operator. Enhancers replace wrappers to take advantage of different optimization opportunities.

A wrapper invokes a logical operator together with the corresponding physical details, *e.g.*, input dataset and schema details (if available). For clarity, I discard such physical details in all the definitions below. In contrast to the logical layer, BigDansing invokes a physical operator for a set $D$ of data units, rather than for a single unit. This enables the processing of multiple units in a single function call. For each logical operator, I define a corresponding wrapper.

(1) **PScope** applies a user-defined selection and projection over a set of data units $D$ and outputs a dataset $D' \subset D$.

\[
P\text{Scope}(D) \rightarrow \{D'\}
\]

(2) **PBlock** takes a dataset $D$ as input and outputs a list of key-value pairs defined by users.

\[
P\text{Block}(D) \rightarrow \text{map}\langle\text{key}, \text{list}\langle U\rangle\rangle
\]

(3) **PIterate** takes a list of lists of $U$s as input and outputs their cross product or a user-defined combination.

\[
P\text{Iterate}(\text{list}\langle\text{list}\langle U\rangle\rangle) \rightarrow \text{list}\langle U\rangle | \text{list}\langle\text{Pair}\langle U\rangle\rangle
\]

(4) **PDetect** receives either a list of data units $U$s or a list of data unit pairs $U$-pairs
and produces a list of violations.

\[ \text{PDetect}(\text{list}\langle U \rangle \mid \text{list}\langle \text{Pair}\langle U \rangle \rangle) \rightarrow \text{list}\langle \text{Violation} \rangle \]

(5) \textbf{PGenFix} receives a list of violations as input and outputs a list of a set of possible fixes, where each set of fixes belongs to a different input violation.

\[ \text{PGenFix}(\text{list}\langle \text{Violation} \rangle) \rightarrow \text{list}\langle \{\text{PossibleFixes}\} \rangle \]

With declarative rules, the operations over a dataset are known, which enables algorithmic opportunities to improve performance. Notice that one could also discover such operations with code analysis over the UDFs [104]. However, I leave this extension to future work. \textsc{BigDansing} exploits such optimization opportunities via three new \texttt{enhancers} operators: \texttt{CoBlock}, \texttt{UCrossProduct}, and \texttt{OJJoin}. \texttt{CoBlock} is a physical operator that allows grouping multiple datasets by a given key. \texttt{UCrossProduct} and \texttt{OJJoin} are basically two additional different implementations for the \texttt{Plterate} operator.

### 3.3.2 From Logical to Physical Plan

As mentioned earlier, optimizing a logical plan is performed by static analysis (plan consolidation) and by plugging \texttt{enhancers} (operators translation) whenever possible.

\textbf{Plan Consolidation.} Whenever logical operators use a different label for the same dataset, \textsc{BigDansing} translates them into distinct physical operators. \textsc{BigDansing} has to create multiple copies of the same dataset, which it might broadcast to multiple nodes. Thus, both the memory footprint at compute nodes and the network traffic are increased. To address this problem, \textsc{BigDansing} consolidates redundant logical operators into a single logical operator. Hence, by applying the same logical operator
Algorithm 1: Logical plan consolidation

```java
input : LogicalPlan lp
output: LogicalPlan clp
1 PlanBuilder lpb = new PlanBuilder();
2 for logical operator lop_i \in lp do
3     lop_j ← findMatchingLO(lop_i, lp);
4     DS_1 ← getSourceDS(lop_i);
5     DS_2 ← getSourceDS(lop_j);
6     if DS_1 == DS_2 then
7         lop_c ← getLabelsFuncs(lop_i, lop_j);
8         lop_c.setInput(DS_1, DS_2);
9         lpb.add(lop_c);
10        lp.remove(lop_i, lop_j);
11    if lpb.hasConsolidatedOps then
12        lpb.add(lp.getOperators());
13        return lpb.generateConsolidatedLP();
14    else
15        return lp;
```

several times on the same set of data units using shared scans, BIGDANSING is able to increase data locality and reduce I/O overhead.

Algorithm 1 details the consolidation process for an input logical plan \( lp \). For each operator \( lop_i \), the algorithm looks for a matching operator \( lop_j \) (Lines 2-3). If \( lop_j \) has the same input dataset as \( lop_i \), BIGDANSING consolidates them into \( lop_c \) (Lines 4-6). The newly consolidated operator \( lop_c \) takes the labels, functions, and datasets from \( lop_i \) and \( lop_j \) (Lines 7-8). Next, BIGDANSING adds \( lop_c \) into a logical plan builder \( lpb \) and removes \( lop_i \) and \( lop_j \) from \( lp \) (Lines 9-10). At last, if any operator was consolidated, it adds the non-consolidated operators to \( lpb \) and returns the consolidated logical plan (Lines 11-13). Otherwise, it returns \( lp \) (Line 15).

Let me now illustrate the consolidation process with an example. Consider a DC on the TPC-H database stating that if a customer and a supplier have the same name and phone, they must be in the same city. Formally,
Figure 3.5: Plans for DC in rule 3.1

\[ \text{DC} : \forall t_1, t_2 \in D1, \neg(t_1.c\text{-}name = t_2.s\text{-}name \land t_1.c\text{-}phone = t_2.s\text{-}phone \land t_1.c\text{-}city \neq t_2.s\text{-}city) \]  

For this DC, **BigDansing** generates the logical plan in Figure 3.5(a) with operators **Scope** and **Block** applied twice over the same input dataset. It then consolidates redundant logical operators into a single one (Figure 3.5(b)), thereby reducing the overhead of reading an input dataset multiple times. The logical plan consolidation is only applied when it does not affect the original labeling of the operators.

**Operators Translation.** Once a logical plan has been consolidated, **BigDansing** translates the consolidated logical plan into a physical plan. It maps each logical operator to its corresponding **wrapper**, which in turn maps to one or more physical operators. For example, it produces the physical plan in Figure 3.5(c) for the consolidated logical plan in Figure 3.5(b). For **enhancers**, **BigDansing** exploits some particular information from the data cleansing process. Below, I detail these three enhancers operators, as well as in which cases they are used by my system.
• *CoBlock* takes multiple input datasets $\mathbb{D}$ and applies a group-by on a given key. This would limit the comparisons required by the rule to only blocks with the same key from the different datasets. An example is shown in Figure A.1 (Appendix A). Similar to CoGroup defined in [55], the output of *CoBlock*, all keys from both inputs are collected into bags. The output of this operator is a map from a key value to the list of data units sharing that key value. I formally define this operator as:

$$ \text{CoBlock} (\mathbb{D}) \rightarrow \text{map}\{\text{key}, \text{list}\langle \text{list}\langle U \rangle \rangle\}$$

If two non-consolidated *Block* operators' outputs go to the same *Iterate*, and then to a single *PDetect*, **BIGDANSING** translates them into a single *CoBlock*. Using *CoBlock* allows me to reduce the number of candidate violations for *Detect*. This is because *Iterate* generates candidates only inside and not across CoBlocks (see Figure 3.6).

• *UCrossProduct* receives a single input dataset $D$ and applies a self cross product over it. This operation is usually performed in cleansing processes that would output the same violations, irrespective of the order of the input of *Detect*. *UCrossProduct* avoids redundant comparisons, reducing the number of comparisons from $n^2$ to $\frac{n \times (n-1)}{2}$, with $n$ being the number of units $U$s in the input. For example, the output of the logical operator *Iterate* in Figure 3.2 is the result of *UCrossProduct* within each block; there are four pairs instead of thirteen, since the operator avoids three comparisons for the elements in block B1 and six for the ones in B3 of Figure 3.2. Formally:

$$ \text{UCrossProduct}(D) \rightarrow \text{list}\langle \text{Pair}\langle U \rangle \rangle$$

If, for a single dataset, the declarative rules contain only symmetric comparisons, *e.g.*, $=$ and $\neq$, then the order in which the tuples are passed to *PDetect* (or to the next logical operator if any) does not matter. In this case, **BIGDANSING** uses *UCrossProduct* to avoid materializing many unnecessary pairs of data units, such as
Figure 3.6: Example plans with CoBlock

for the Iterate operator in Figure 3.2. It also uses UCrossProduct when: (i) users do not provide a matching Block operator for the Iterate operator; or (ii) users do not provide any Iterate or Block operator.

- OCJoins performs a self join on one or more ordering comparisons (i.e., $<, >, \geq, \leq$). This is a very common operation in rules such as DCs. Thus, BIGDANSING provides OCJoin, which is an efficient operator to deal with join ordering comparisons. It takes input dataset $D$ and applies a number of join conditions, returning a list of joining $U$-pairs. I formally define this operator as follows:

$$\text{OCJoin}(D) \rightarrow \text{list}\langle\text{Pair}\langle U\rangle\rangle$$

Every time BIGDANSING recognizes joins conditions defined with ordering comparisons in PDetect, e.g., $\phi_D$, it translates Iterate into a OCJoin implementation. Then, it passes the OCJoin output to a PDetect operator (or to the next logical operator if any).
3.3.3 Fast Joins with Ordering Comparisons

Existing systems handle joins over ordering comparisons using a cross product and a post-selection predicate, leading to poor performance. **BigDansing** provides an efficient ad-hoc join operator, referred to as OCJoin, to handle these cases. The main goal of OCJoin is to increase the ability to process joins over ordering comparisons in parallel, and to reduce its complexity by reducing the algorithm’s search space. In a nutshell, OCJoin first range partitions a set of data units and sorts each of the resulting partitions in order to validate the inequality join conditions in a distributed fashion. OCJoin works in four main phases: partitioning, sorting, pruning, and joining (see Algorithm 2).

**Partitioning.** OCJoin first selects the attribute, *PartAtt*, on which to partition the input $D$ (line 1). I assume that all join conditions have the same output cardinality. This can be improved using cardinality estimation techniques [15,57], but it is beyond the scope of the chapter. OCJoin chooses the first attribute involved in the first condition. For instance, consider again $\phi_D$ (Example 1), OCJoin sets *PartAtt* to *rate* attribute. Then, OCJoin partitions the input dataset $D$ into $nbParts$ range partitions based on *PartAtt* (line 2). As part of this partitioning, OCJoin distributes the resulting partitions across all available computing nodes. Notice that OCJoin runs the range partitioning in parallel. Next, OCJoin forks a parallel process for each range partition $k_i$ to run the three remaining phases (lines 3-14).

**Sorting.** For each partition, OCJoin creates as many sorting lists (*Sorts*) as inequality conditions are in a rule (lines 4-5). For example, OCJoin creates two sorted lists for $\phi_D$: one sorted on *rate* and the other sorted on *salary*. Each list contains the attribute values on which the sort order is and the tuple identifiers. Note that OCJoin only performs a local sorting in this phase and hence it does not require any data transfer across nodes. Since multiple copies of a partition may exist in multiple computing
Algorithm 2: OCJoin

\[\begin{align*}
\textbf{input :} & \text{ Dataset } D, \text{ Condition } \text{conds}[] , \text{ Integer } \text{nbParts} \\
\textbf{output:} & \text{ List Tuples(Tuple)} \\
\end{align*}\]

// PARTITIONING PHASE

1. PartAtt ← getPrimaryAtt(conds[].getAttribute());
2. \(K \leftarrow \text{RangePartition}(D, \text{PartAtt}, \text{nbParts})\);
3. \(\text{for each } k_i \in K \text{ do} \)
   \(\text{for each } c_j \in \text{conds}[] \text{ do} \) // SORTING
   \(\text{Sorts}[j] \leftarrow \text{sort}(k_i, c_j.\text{getAttribute})();\)
   \(\text{for each } k_l \in \{k_i+1...k|K|\} \text{ do} \)
5. \(\text{if } \text{overlap}(k_i, k_l, \text{PartAtt}) \text{ then} \) // PRUNING
6. \(\text{tuples} = \emptyset;\)
   \(\text{for each } c_j \in \text{conds}[] \text{ do} \) // JOINING
7. \(\text{tuples} \leftarrow \text{join}(k_i, k_l, \text{Sorts}[j], \text{tuples});\)
8. \(\text{if } \text{tuples} == \emptyset \text{ then} \)
   \(\text{break;} ;\)
9. \(\text{if } \text{tuples} != \emptyset \text{ then} \)
10. \(\text{Tuples.add(tuples);}\)

nodes, I apply sorting before pruning and joining phases to ensure that each partition is sorted at most once.

**Pruning.** Once all partitions \(k_i \in K\) are internally sorted, OCJoin can start joining each of these partitions based on the inequality join conditions. However, this would require transferring large amounts of data from one node to another. To circumvent such an overhead, OCJoin inspects the min and max values of each partition to avoid joining partitions that do not overlap in their min and max range (the pruning phase, line 7). Non-overlapping partitions do not produce any join result. If the selectivity values for the different inequality conditions are known, OCJoin can order the different joins accordingly.

**Joining.** OCJoin finally proceeds to join the overlapping partitions and outputs the join results (lines 9-14). For this, it applies a distributed sort merge join over the sorted lists, where some partitions are broadcast to other machines while keeping
the rest locally. Through pruning, OCJoin tells the underlying distributed processing platform which partitions to join. It is up to that platform to select the best approach to minimize the number of broadcast partitions.

### 3.3.4 Translation to Spark Execution Plans

Spark represents datasets as a set of Resilient Distributed Datasets (RDDs), where each RDD stores all Us of an input dataset in sequence. Thus, the Executor represents each physical operator as an RDD data transformation. I implement the full stack translation for all physical operators in Spark since it is my main production system. I show in Figure 3.7 a summary of BigDANSING’s operators for Spark.

**Spark-PScope.** The Executor receives a set of RDDs as well as a set of PScope operators. It links each RDD with one or more PScope operators, according to their labels. Then, it simply translates each PScope to a map() Spark operation over its RDD. Spark, in turn, takes care of automatically parallelizing the map() operation over all input Us. As a PScope might output a null or empty U, the Executor applies a filter() Spark operation to remove null and empty Us before passing them to the next operator.
Spark-PBlock. The Executor applies one Spark `groupBy()` operation for each PBlock operator over a single RDD. BigDANSING automatically extracts the key from each $U$ in parallel and passes it to Spark, which in turn uses the extracted key for its `groupBy` operation. As a result, Spark generates an RDDPair (a key-value pair data structure) containing each a grouping key (the key in the RDDPair) together with the list of all $Us$ sharing the same key (the value in the RDDPair).

Spark-CoBlock. The Executor receives a set of RDDs and a set of PBlock operators with matching labels. Similar to the Spark-PBlock, the Spark-CoBlock groups each input RDD (with `groupBy()`) using its corresponding PBlock. In addition, it performs a `join()` Spark operation on the keys of the output produced by `groupBy()`. Spark-CoBlock also outputs an RDDPair, but in contrast to Spark-PBlock, the produced value is a set of lists of $Us$ from all input RDDs sharing the same extracted key.

Spark-CrossProduct & -UCrossProduct. The Executor receives two input RDDs and outputs an RDDPair of the resulting cross product. Notice that I extended Spark’s Scala code with a new function `selfCartesian()` in order to efficiently support the UCrossProduct operator. Basically, `selfCartesian()` computes all the possible combinations of pair-tuples in the input RDDs.

Spark-OCJoin. The Executor receives two RDDs and a set of inequality join conditions as input. The Executor applies the OCJoin operator on top of Spark as follows. First, it extracts PartAtt (the attribute on which it has to partition the two input RDDs) from both RDDs by using the `keyBy()` Spark function. Then, the Executor uses the `sortByKey()` Spark function to perform a range partitioning of both RDDs. As a result, the Executor produces a single RDD containing several data blocks using the `mapPartitions()` Spark function. Each data block provides as many lists as inequality join conditions; each containing all $Us$ sorted on a different attribute involved in the join conditions. Finally, the Executor uses the `selfCartesian()` Spark function
to generate unique sets of paired data blocks.

**Spark-PDetect.** This operator receives a **Plterate** operator, a **PDetect** operator, and a single RDD as input. The **Executor** first applies the **Plterate** operator and then the **PDetect** operator on the output. The **Executor** implements this operator using the `map()` Spark function.

**Spark-PGenFix** The **Executor** applies a **PGenFix** on each input RDD using spark’s `map()` function. When processing multiple rules on the same input dataset, the **Executor** generates an independent RDD of fixes for each rule. After that it combines all RDDs of possible repairs into a single RDD and passes it to **BigDansing**’s repair algorithm. This operator is also implemented by the **Executor** inside the **Detect** operator for performance optimization purposes.

### 3.3.5 Translation to MapReduce Execution Plans

I now briefly describe how the **Executor** runs the five wrapper physical operators on MapReduce. Since MapReduce is only a proof of concept for **BigDansing**, I am not considering in this chapter the translation of *enhancers*.

**MR-PScope.** The **Executor** translates the **PScope** operator into a **Map** task whose `map` function applies the received **PScope**. Null and empty U are discarded within the same **Map** task before passing them to the next operator.

**MR-PBlock.** The **Executor** translates the **PBlock** operator into a **Map** task whose `partitioner` function applies the received **PBlock** to set the intermediate key. The MapReduce framework automatically groups all Us that share the same key. The **Executor** does the same for the **CoBlock** operator, but it also labels each intermediate key-value pair with the input dataset label for identification at **Reduce** tasks.

**MR-PIterate.** The **Executor** translates **Plterate** into a **Reduce** task whose `reduce` function applies the received **Plterate**.
MR-PDetect. The Executor translates the PDetect operator into a Reduce task whose reduce function applies the received PDetect. The Executor might also apply the received PDetect in the reduce function of a Combine task.

MR-PGenFix. The Executor translates the PGenFix operator into a Map task whose map function applies the received PRepair. The Executor might also apply the received PGenFix at the reduce function of PDetect.

3.3.6 Physical Data Access

BigDansing applies three different data storage optimizations for Spark’s and MapReduce input: (i) data partitioning to avoid shuffling large amounts of data; (ii) data replication to efficiently support a large variety of data quality rules; and (iii) data layouts to improve I/O operations. I describe them below.

(1) Partitioning. Typically, distributed data storage systems split data files into smaller chunks based on size. In contrast, BigDansing partitions a dataset based on its content, i.e., based on attribute values. Such a logical partitioning allows to co-locate data based on a given blocking key. As a result, BigDansing can push down the Block operator to the storage manager. This allows avoiding to co-locate datasets while detecting violations and hence significantly reduce the network costs.

(2) Replication. A single data partitioning, however, might not be useful for multiple data cleansing tasks. In practice, I may need to run several data cleansing jobs as data cleansing tasks do not share the same blocking key. To handle such a case, I replicate a dataset in a heterogenous manner. In other words, BigDansing logically partitions (i.e., based on values) each replica on a different attribute. As a result, I can again push down the Block operator for multiple data cleansing tasks.

(3) Layout. BigDansing converts a dataset to a binary format when storing it in the underlying data storage framework. This helps avoid expensive string parsing
operations. Also, in most cases, binary format ends up reducing the file size and hence I/Os. Additionally, I store a dataset in a column-oriented fashion. This enables pushing down the **Scope** operator to the storage manager and hence reduces I/O costs significantly.

For the underlying data storage layer, I use Cartilage to store data and access data from HDFS. Cartilage works both with Hadoop and Spark and uses HDFS as the underlying distributed file system. Using Cartilage, the storage manager essentially translates **BigDansing** data access operations (including the operator pushdowns) to three basic HDFS data access operations: (i) **Path Filter**, to filter the input file paths; (ii) **Input Format**, to assign input files to workers; and (iii) **Record Reader**, to parse the files into tuples. In Spark, this means that I specify these three UDFs when creating RDDs. As a result, I can manipulate the data access right from HDFS so that this data access is invisible and completely non-invasive to Spark. To leverage all the data storage optimizations done by **BigDansing**, I indeed need to know how the data was uploaded in the first place, e.g., in which layout and sort order the data is stored. To allow **BigDansing** to know so, in addition to datasets, I store the *upload plan* of each uploaded dataset, which is essentially the upload metadata. At query time, **BigDansing** uses this metadata to decide how to access an input dataset, e.g., if it performs a full scan or an index scan, using the right UDF (path filter, input format, record reader) implementation.

### 3.4 Distributed Repair Algorithms

Most of the existing repair techniques are centralized. I present two approaches to implement a repair algorithm in **BigDansing**. First, I show how my system can run a centralized data repair algorithm in parallel, without changing the algorithm. In other words, **BigDansing** treats that algorithm as a black box. Second, I design a distributed version of the widely used *equivalence class* algorithm.
3.4.1 Scaling Data Repair as a Black Box

Overall, I divide a repair task into independent smaller repair tasks. For this, I represent the violation graph as a hypergraph containing the violations and their possible fixes. The nodes represent the elements and each hyperedge covers a set of elements that together violate a rule, along with possible repairs. I then divide the hypergraph into smaller independent subgraphs, i.e., connected components, and I pass each connected component to an independent data repair instance. Note that the hypergraph representation allows BigDansing to model logically implicated violations when processing multiple rules simultaneously. However, it is the responsibility of the repair algorithm to handle them.

**Connected components.** Given an input set of violations, from at least one rule, BigDansing first creates their hypergraph representation of such possible fixes in a distributed manner. It then uses the Bulk Synchronous Parallel (BSP) graph processing model [20] to find all connected components in the hypergraph in parallel. As a result, BigDansing obtains a connected component ID for each independent
hyperedge. It then groups hyperedges by the connected component ID. Figure 3.8 shows an example of connected components in a hypergraph containing three violations $v_1$, $v_2$, and $v_3$. Note that violations $v_1$ and $v_2$ can be caused by different rules. Violations $v_1$ and $v_2$ are grouped in a single connected component $CC_1$, because they share element $c_2$. In contrast, $v_3$ is assigned to a different connected component $CC_2$, because it does not share any element with $v_1$ or $v_2$.

**Processing framework for hypergraphs.** The selection of the distributed graph processing system depends on the size of the violation hypergraph. By default, BigDansing uses GraphX to find all connected components of the violation hypergraph in parallel. GraphX is designed to act as a distributed graph processing system on top of Spark. As a result, BigDansing is able to exchange the hypergraph’s information with GraphX seamlessly by utilizing the unified data model of Spark. If the hypergraph is too big (has a large number of vertices and edges) or too dense (highly connected hypergraph), GraphX fails to process the connected components. This is because the higher the data dependency in the graph, the larger the amount of data shuffling is required, which timeouts the connections of the workers in GraphX. In this case, BigDansing utilizes Giraph [29] as an alternative solution, where BigDansing exchanges the violation hypergraph and the result of the connected components with Giraph through HDFS. Giraph is a much more efficient graph processing system compared to GraphX [106], which allows BigDansing to process larger violation hypergraph. However, it adds a huge overhead to BigDansing, where the time required for reading and writing the violation hypergraph into HDFS is huge.

There exist other technologies that are faster than HDFS, such as HDFS on RAM disks (tmpfs or ramfs), in memory distributed file system (Alluxio [107], formerly Tachyon [108]), or by using the cross-platform data flow pipes of Rheem [109,110]. However, in our implementation, we only use HDFS since such technologies are still experimental and out of the scope of this dissertation.
**Independent data repair instance.** Once all connected components are computed, BigDansing assigns each of them to an independent data repair instance and runs such repair instances in a distributed manner (right-side of Figure 3.8). When all data repair instances generate the required fixes, BigDansing updates the input dataset and passes it to the RuleEngine to detect potential violations introduced by the RepairAlgorithm. The number of iterations required to fix all violations depends on the input rules, the dataset, and the repair algorithm.

**Dealing with big connected components.** If a connected component does not fit in memory, BigDansing uses a $k$-way multilevel hypergraph partitioning algorithm [111] to divide it into $k$ equal parts and run them on distinct machines. Unfortunately, naively performing this process can lead to inconsistencies and contradictory choices in the repair. Moreover, it can fix the same violation independently in two machines, thus introducing unnecessary changes to the repair. I illustrate this problem with an example.

**Example 2:** Consider a relational schema $D$ with 3 attributes $A$, $B$, $C$ and 2 FDs $A \rightarrow B$ and $C \rightarrow B$. Given the instance $[t_1](a_1, b_1, c_1), [t_2](a_1, b_2, c_1)$, all data values are in violation: $t_1.A, t_1.B, t_2.A, t_2.B$ for the first FD and $t_1.B, t_1.C, t_2.B, t_2.C$ for the second one. Assuming the compute nodes have enough memory only for five values, I need to solve the violations by executing two instances of the algorithm on two different nodes. Regardless of the selected tuple, suppose the first compute node repairs a value on attribute $A$, and the second one a value on attribute $C$. When I put the two repairs together and check their consistency, the updated instance is a valid solution, but the repair is not minimal because a single update on attribute $B$ would have solved both violations. However, if the first node fixes $t_1.B$ by assigning value “b2” and the second one fixes $t_2.B$ with “b1”, not only there are two changes, but the final instance is also inconsistent. □

I tackle the above problem by assigning the role of master to one machine and the
role of slave to the rest. Every machine applies a repair in isolation, but I introduce an extra test in the union of the results. For the violations that are solved by the master, I mark its changes as immutable, which prevents further changes to a repaired element. If a change proposed by a slave contradicts a possible repair that involve a master’s change, the slave repair is undone and a new iteration is triggered. As a result, the algorithm always reaches a fix point to produce a clean dataset, because an updated value cannot change in the following iterations.

BigDansing currently provides two repair algorithms using this approach: the equivalence class algorithm and a general hypergraph-based algorithm [6, 9]. Users can also implement their own repair algorithm if it is compatible with BigDansing’s repair interface.

3.4.2 Scalable Equivalence Class Algorithm

The idea of the equivalence class based algorithm [59] is to first group all elements that should be equivalent together, and to then decide how to assign values to each group. An equivalence class consists of pairs of the form (t, A), where t is a data unit and A is an element. In a dataset D, each unit t and each element 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). That is, for all (t, A) ∈ E, t[A] has the same value targ(E). The algorithm selects the target value for each equivalence class to obtain a repair with the minimum overall cost.

I extend the equivalence class algorithm to a distributed setting by modeling it as a distributed word counting algorithm based on map and reduce functions. However, in contrast to a standard word count algorithm, I use two map-reduce sequences. The first map function maps the violations’ possible fixes for each connected component into key-value pairs of the form ⟨⟨ccID,value⟩,count⟩. The key ⟨ccID,value⟩ is a composite key that contains the connected component ID and the element value for each
possible fix. The value *count* represents the frequency of the element value, which I initialize to 1. The first *reduce* function counts the occurrences of the key-value pairs that share the same connected component ID and element value. It outputs key-value pairs of the form \( \langle \text{ccID}, \text{value} \rangle, \text{count} \). Note that if an element exists in multiple fixes, I only count its value once. After this first *map-reduce* sequence, another *map* function takes the output of the *reduce* function to create new key-value pairs of the form \( \langle \text{ccID}, \langle \text{value}, \text{count} \rangle \rangle \). The *key* is the connected component ID and the *value* is the frequency of each element value. The last *reduce* selects the element value with the highest frequency to be assigned to all the elements in the connected component *ccID*.

### 3.5 Experimental Study

I evaluate BigDansing using both real and synthetic datasets with various rules. I consider a variety of scenarios to evaluate the system and answer the following questions: (i) how well does it perform compared with baseline systems in a single node setting? (ii) how well does it scale to different dataset sizes compared to the state-of-the-art distributed systems? (iii) how well does it scale in terms of the number of nodes? (iv) how well does its abstraction support a variety of data cleansing tasks, e.g., for deduplication? and (v) how do its different techniques improve performance and what is its repair accuracy?

#### 3.5.1 Setup

Table 3.1 summarizes the datasets and Table 3.2 shows the rules I use for my experiments.

*(1) TaxA.* Represents personal tax information in the US \[4\]. Each row contains a person’s name, contact information, and tax information. For this dataset, I use the FD rule \( \varphi_1 \) in Table 3.2. I introduced errors by adding random text to attributes City
Table 3.1: Statistics of the datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rows</th>
<th>Dataset</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaxA&lt;sub&gt;1&lt;/sub&gt;–TaxA&lt;sub&gt;5&lt;/sub&gt;</td>
<td>100K – 40M</td>
<td>customer2</td>
<td>32M</td>
</tr>
<tr>
<td>TaxB&lt;sub&gt;1&lt;/sub&gt;–TaxB&lt;sub&gt;3&lt;/sub&gt;</td>
<td>100K – 3M</td>
<td>NCVoter</td>
<td>9M</td>
</tr>
<tr>
<td>TPCH&lt;sub&gt;1&lt;/sub&gt;–TPCH&lt;sub&gt;10&lt;/sub&gt;</td>
<td>100K – 1907M</td>
<td>HAI</td>
<td>166k</td>
</tr>
<tr>
<td></td>
<td></td>
<td>customer1</td>
<td>19M</td>
</tr>
</tbody>
</table>

Table 3.2: Integrity constraints used for testing

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varphi_1) (FD):</td>
<td>Zipcode (\rightarrow) City</td>
</tr>
<tr>
<td>(\varphi_2) (DC):</td>
<td>(\forall t_1, t_2 \in TaxB, \neg(t_1.\text{Salary} &gt; t_2.\text{Salary} \land t_1.\text{Rate} &lt; t_2.\text{Rate}))</td>
</tr>
<tr>
<td>(\varphi_3) (FD):</td>
<td>o_custkey (\rightarrow) c_address</td>
</tr>
<tr>
<td>(\varphi_4) (UDF):</td>
<td>Two rows in Customer are duplicates</td>
</tr>
<tr>
<td>(\varphi_5) (UDF):</td>
<td>Two rows in NCVoter are duplicates</td>
</tr>
<tr>
<td>(\varphi_6) (FD):</td>
<td>Zipcode (\rightarrow) State</td>
</tr>
<tr>
<td>(\varphi_7) (FD):</td>
<td>PhoneNumber (\rightarrow) Zipcode</td>
</tr>
<tr>
<td>(\varphi_8) (FD):</td>
<td>ProviderID (\rightarrow) City, PhoneNumber</td>
</tr>
</tbody>
</table>

and State at a 10% rate.

(2) **TaxB.** I generate TaxB by adding 10% numerical random errors on the Rate attribute of TaxA. My goal is to validate the efficiency with rules that have inequality conditions only, such as DC \(\varphi_2\) in Table 3.2.

(3) **TPCH.** From the TPC-H benchmark data [112], I joined the lineitem and customer tables and applied 10% random errors on the address. I use this dataset to test FD rule \(\varphi_3\).

(4) **Customer.** In my deduplication experiment, I use TPC-H customer with 4.5 million rows to generate tables customer1 with 3x exact duplicates and customer2 with 5x exact duplicates. Then, I randomly select 20% of the total number of tuples, in both relations, and duplicate them with random edits on attributes name and phone.

(5) **NCVoter.** This is a real dataset that contains North Carolina voter information. I added 20% random duplicate rows with random edits in name and phone attributes.

This real dataset contains hospital information and statistics measurements for infections developed during treatment. I added 10% random errors on the attributes covered by the FDs and tested four combinations of rules ($\varphi_6 - \varphi_8$ from Table 3.2). Each rule combination has its own dirty dataset.

To my knowledge, there exists only one full-fledged data cleansing system that can support all the rules in Table 3.2.

(1) NADEEF [5]: An open-source single-node platform supporting both declarative and user-defined quality rules.

(2) PostgreSQL v9.3: Since declarative quality rules can be represented as SQL queries, I also compare to PostgreSQL for violation detection. To maximize benefits from large main memory, I configured it using pgtune [113].

The two declarative constraints, DC $\varphi_2$ and FD $\varphi_3$ in Table 3.2 are translated to SQL as shown below.

\[
\varphi_2: \quad \text{SELECT } a.\text{Salary}, b.\text{Salary}, a.\text{Rate}, b.\text{Rate} \\
\text{FROM } \text{TaxB } a \text{ JOIN TaxB } b \\
\text{WHERE } a.\text{Salary} > b.\text{Salary} \text{ AND } a.\text{Rate} < b.\text{Rate};
\]

\[
\varphi_3: \quad \text{SELECT } a.\text{o_custkey}, a.\text{c_address}, b.\text{c_address} \text{ FROM } \\
\text{TPCH } a \text{ JOIN TPCH } b \text{ ON } a.\text{custkey} = b.\text{custkey} \\
\text{WHERE } a.\text{c_address} \neq b.\text{c_address};
\]

I also consider two parallel data processing frameworks:

(3) Shark 0.8.0 [114]: This is a scalable data processing engine for fast data analysis. I selected Shark due to its scalability advantage over existing distributed SQL engines.

(4) Spark SQL v1.0.2: It is an experimental extension of Spark v1.0 that allows users to run relational queries natively on Spark using SQL or HiveQL. I selected this system as, like BigDancing, it runs natively on top of Spark.
I ran my experiments in two different settings: (i) a single-node setting using a Dell Precision T7500 with two 64-bit quad-core Intel Xeon X5550 (8 physical cores and 16 CPU threads) and 58GB RAM and (ii) a multi-node setting using a compute cluster of 17 Shuttle SH55J2 machines (1 master with 16 workers) equipped with Intel i5 processors with 16GB RAM constructed as a star network.

### 3.5.2 Single-Node Experiments

For these experiments, I start comparing BigDansing with NADEEF in the execution times of the whole cleansing process (i.e., detection and repair). Figure 3.9(a) shows the performance of both systems using the TaxA and TPCH datasets with 100K and 1M (200K for ϕ₂) rows. I observe that BigDansing is more than three orders of magnitude faster than NADEEF in rules ϕ₁ (1M) and ϕ₂ (200K), and ϕ₃ (1M). In fact, NADEEF is only “competitive” in rule ϕ₁ (100K), where BigDansing is only twice faster. The high superiority of BigDansing comes from two main reasons: (i) In contrast to NADEEF, it provides a finer granular abstraction allowing users to specify rules more efficiently; and (ii) It performs rules with inequality conditions in an efficient way (using OCJoin). In addition, NADEEF issues thousands
of SQL queries to the underlying DBMS for detecting violations. I do not report results for larger datasets because NADEEF was not able to run the repair process for more than 1M rows (300K for $\varphi_2$).

I also observed that violation detection was dominating the entire data cleansing process. Thus, I ran an experiment for rule $\varphi_1$ in TaxA with 1M rows by varying the error rate. Violation detection takes more than 90% of the time, regardless of the error rate (Figure 3.9(b)). In particular, I observe that even for a very high error rate (50%), the violation detection phase still dominates the cleansing process. Notice that for more complex rules, such as rule $\varphi_2$, the dominance of the violation detection is even
amplified. Therefore, to be able to extensively evaluate BigDansing, I continue my experiments focusing on the violation detection phase only, except if stated otherwise.

Figures 3.10(a), Figure 3.10(b), and 3.10(c) show the violation detection performance of all systems for TaxA, TaxB, and TPCH datasets. TaxA and TPCH datasets have 100K, 1M, and 10M rows and TaxB has 100K, 200K, and 300K rows. With equality-based rules $\varphi_1$ and $\varphi_3$, I observe that PostgreSQL is always faster than all other systems on the small datasets (100K rows). However, once I increase the size by one order of magnitude (1M rows), I notice the advantage of BigDansing: it is at least twice faster than PostgreSQL and more than one order of magnitude faster than
NADEEF. For the largest dataset (10M rows), BIGDANSING is almost two orders of magnitude faster than PostgreSQL for the FD in TaxA and one order of magnitude faster than PostgreSQL for the DC in TaxA. For the FD in TPCH, BIGDANSING is twice faster than PostgreSQL. It is more than three orders of magnitude faster than NADEEF for all rules. Overall, I observe that BIGDANSING performs similarly to Spark SQL. The small difference is that Spark SQL uses multithreading better than BIGDANSING. I can explain the superiority of BIGDANSING compared to the other systems along two reasons: (i) BIGDANSING reads the input dataset only once, while PostgreSQL and Shark read it twice because of the self joins; and (ii) BIGDANSING does not generate duplicate violations, while SQL engines do when comparing tuples using self-joins, such as for TaxA and TPCH’s FD. Concerning the inequality-based rule $\phi_2$, I limited the runtime to four hours for all systems. BIGDANSING is one order of magnitude faster than all other systems for 100K rows. For 200K and 300K rows, it is at least two orders of magnitude faster than all baseline systems. Such performance superiority is achieved by leveraging the inequality join optimization that is not supported by the baseline systems.

3.5.3 Multi-Node Experiments

I now compare BIGDANSING with Spark SQL and Shark in the multi-node setting. I also implemented a lighter version of BIGDANSING on top of Hadoop MapReduce to show BIGDANSING independence w.r.t. the underlying framework. I set the size of TaxA to 10M, 20M, and 40M rows. Moreover, I tested the inequality DC $\varphi_2$ on TaxB dataset with sizes of 1M, 2M, and 3M rows. I limited the runtime to 40 hours for all systems.

BIGDANSING-Spark is slightly faster than Spark SQL for the equality rules in Figure 3.11(a). Even though BIGDANSING-Spark and Shark are both implemented on top of Spark, BIGDANSING-Spark is up to three orders of magnitude faster than
Figure 3.12: Results for the experiments on (a) scale-out, (b) deduplication, and (c) physical optimizations.

Shark. Even BigDansing-Hadoop is doing better than Shark (Figure 3.11(a)). This is because Shark does not process joins efficiently. The performance of BigDansing over baseline systems is magnified when dealing with inequalities. I observe that BigDansing-Spark is at least two orders of magnitude faster than both Spark SQL and Shark (Figure 3.11(b)). I had to stop Spark SQL and Shark executions after 40 hours of runtime; both Spark SQL and Shark are unable to process the inequality DC efficiently.

I also included a testing for large TPCH datasets of sizes 150GB, 200GB, 250GB, and 300GB (959M, 1271M, 1583M, and 1907M rows resp.) producing between 6.9B...
and 13B violations. I excluded Shark as it could not run on these larger datasets. **BigDansing-**Spark is 16 to 22 times faster than **BigDansing-Hadoop** and 6 to 8 times faster than Spark SQL (Figure 3.11(c)). The performance difference between **BigDansing-**Spark and **BigDansing-Hadoop** stems from Spark being generally faster than Hadoop; Spark is an in-memory data processing system while Hadoop is disk-based. Furthermore, **BigDansing-**Spark significantly outperforms Spark SQL because **BigDansing** has a better execution plan for $\varphi_3$ compared to Spark SQL. In particular, the execution plan of **BigDansing** eliminates duplicate results earlier than Spark SQL’s plan, which reduces the memory and network overheads of **BigDansing-**Spark. Moreover, unlike Spark SQL, **BigDansing-**Spark has a lower I/O complexity due to its optimized physical data access.

### 3.5.4 Scaling BigDansing Out

I compared the speedup of **BigDansing-**Spark to Spark SQL when increasing the number of workers on a dataset size of 500M rows. I observe that **BigDansing-**Spark is at least 3 times faster than Spark SQL starting from one single worker and up to 16 (Figure 3.12(a)). I also notice that **BigDansing-**Spark is about 1.5 times faster than Spark SQL while using only half the number of workers used by Spark SQL. Although both **BigDansing-**Spark and Spark SQL generally have a good scalability, **BigDansing-**Spark performs better than Spark SQL on large input datasets because the execution plan of **BigDansing-**Spark does not copy the input data twice.

### 3.5.5 Deduplication in BigDansing

I show that one can run a deduplication task with **BigDansing**. I use *cust1*, *cust2*, and *NCVoters* datasets on my compute cluster. I implemented a Java version of the Levenshtein distance and use it as a UDF in both **BigDansing** and Shark. Note that I do not consider Spark SQL in this experiment since UDFs cannot be implemented.
directly within Spark SQL. That is, to implement a UDF in Spark SQL the user has to either use a Hive interface or apply a post processing step on the query result. Figure 3.12(b) shows the results. I observe that BigDansing outperforms Shark for both small datasets as well as large datasets. In particular, I see that for cust2 BigDansing outperforms Shark up to an improvement factor of 67. These results not only show the generality of BigDansing supporting a deduplication task, but also the high efficiency of my system.

3.5.6 BigDansing In-Depth

Physical optimizations. I first focus on showing the benefits in performance of the UCrossProduct and OCJoin operators. I use the second inequality DC in Table 3.2 with TaxB dataset on my multi-node cluster. Figure 3.12(c) reports the results of this experiment. I notice that the UCrossProduct operator has a slight performance advantage compared to the CrossProduct operator. This performance difference increases with the dataset size. However, by using the OCJoin operator, BigDansing becomes more than two orders of magnitudes faster compared to both cross product operators (up to an improvement factor of 655).
**Abstraction advantage.** I now study the benefits of **BIGDANSING**’s abstraction. I consider the deduplication scenario, in Chapter 3.5.5, with the smallest **TaxA** dataset on my single-node machine. I compare the performance difference between **BIGDANSING** using its full API, and **BIGDANSING** using only the **Detect** operator. I see in Figure 3.13(a) that running a UDF using the full **BIGDANSING** API makes **BIGDANSING** three orders of magnitudes faster compared to using **Detect** only. This clearly shows the benefits of the five logical operators, even for single-node settings.

**Scalable data repair.** I study the runtime efficiency of the repair algorithms used by **BIGDANSING**. I ran an experiment for rule $\varphi_1$ in **TaxA** with 1M rows by varying the error rate and considering two versions of **BIGDANSING**: one with the parallel data repair, and a baseline with a centralized data repair, such as in **NADEEF**. Figure 3.13(b) shows the results. The parallel version outperforms the centralized one, except when the error rate is very small (1%). For higher rates, **BIGDANSING** is clearly faster, since the number of connected components to deal with in the repair process increases with the number of violations and thus the parallelization provides a stronger boost. Naturally, my system scales much better with the number of violations.

**Repair accuracy.** I evaluate the accuracy of **BIGDANSING** using the traditional precision and recall measure: precision is the ratio of correctly updated attributes (exact matches) to the total number of updates; recall is the ratio of correctly updated attributes to the total number of errors.

I test **BIGDANSING** with the equivalence class algorithm using **HAI** on the following rule combinations: (a) FD $\varphi_6$; (b) FD $\varphi_6$ and FD $\varphi_7$; (c) FD $\varphi_6$, FD $\varphi_7$, and FD $\varphi_8$. Notice that **BIGDANSING** runs (a)-combination alone while it runs (b)-combination and (c)-combination concurrently.

Table 3.3 shows the results for the equivalence class algorithm in **BIGDANSING** and **NADEEF**. I observe that **BIGDANSING** achieves similar accuracy and recall as the one obtained in a centralized system, i.e., **NADEEF**. In particular, **BIGDANSING** requires
<table>
<thead>
<tr>
<th>Rule(s)</th>
<th>NADEEF precision</th>
<th>NADEEF recall</th>
<th>BigDansing precision</th>
<th>BigDansing recall</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.713</td>
<td>0.776</td>
<td>0.714</td>
<td>0.777</td>
<td>1</td>
</tr>
<tr>
<td>6 &amp; 7</td>
<td>0.861</td>
<td>0.875</td>
<td>0.861</td>
<td>0.875</td>
<td>2</td>
</tr>
<tr>
<td>6, 7 &amp; 8</td>
<td>0.923</td>
<td>0.928</td>
<td>0.924</td>
<td>0.929</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R, G</td>
<td>e</td>
</tr>
<tr>
<td>φD</td>
<td>17.1</td>
<td>8183</td>
<td>17.1</td>
<td>8221</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.3: Repair quality using HAI and TaxB datasets.

I also test BigDansing with the hypergraph algorithm using DC rule φD on a TaxB dataset. As the search space of the possible solutions in φD is huge, the hypergraph algorithm uses quadratic programming to approximate the repairs in φD [6]. Thus, I use the Euclidean distance to measure the repairs accuracy on the repaired data attributes compared to the attributes of the ground truth.

Table 3.3 shows the results for the hypergraph algorithm in BigDansing and NADEEF. Overall, I observe that BigDansing achieves the same average distance (|R, G|/e) and similar total distance (|R, G|) between the repaired data R and the ground truth G. Again, BigDansing requires the same number of iterations as NADEEF to completely repair the input dataset. These results confirm that BigDansing achieves the same data repair quality as in a single node setting. This by providing better data cleansing runtimes and scalability than baselines systems.
3.6 Use Case: Violation Detection in RDF

Resource Description Framework (RDF) is a fast growing data model that is used by large public knowledge bases, such as Bio2RDF [115], DBpedia [116], Probase [117], PubChemRDF [118], and UniProt [119], to express and interchange facts on the semantic web. These independent datasets are automatically extracted and heavily interlinked, which make them prone to errors and inconstancies. In this chapter, I explain the type of mistakes that RDF data may contain and explain how BigDansing works for RDF data cleansing on an example error and dirty dataset. Note that it is not intended to present BigDansing as an efficient RDF query engine, such as AdPart [120, 121], S2RDF [122], and Trinity.RDF [123]. Instead, I am only showing the generality of BigDansing and its ability to handle schema-less data, such as RDF, efficiently.

3.6.1 Source of Errors

Each RDF dataset consists of triples in the form \((\text{subject}, \text{predicate}, \text{object})\). Each triple represents a directed labeled edge in the RDF graph from a subject vertex to an object vertex. The predicate is the edge label that describes the relationship between the subject and the object in the RDF triple.

There are six types of errors in RDF data [124], which may affect the nodes or the edge label individually, or influence the whole RDF triple. 

(i) Typos; they are errors in the subjects, objects or predicates that are caused by either human error or from an invalid data extraction. The connectivity of the RDF graph may change if typos affected a subject or an object. (ii) Polymorphism, which is caused by having multiple representations to the same entity. They are similar to typos; however, they are not considered as wrong or misspelled entries. For example, \((\text{Robert, type, professor})\) and \((\text{William, type, prof.})\) are both of the same type, i.e., they have the same RDF predicate and object, but the object is represented differ-
ently. (iii) Missing data; they are either missing values within the RDF data, i.e., by having a Null value in the object, or by lacking crucial triples. (iv) Entities mismatching; they exist when similar RDF entities contain conflicting values, i.e., two RDF triples for the same person that have types Student and Professor simultaneously. (v) Outdated entities; they are RDF triplets that have old or logically incorrect information, such as disconnected phone numbers or invalid addresses for individuals. (vi) Integrity violations; they are caused by inconsistencies in RDF data that violates a business rule or a pre-defined integrity constraint. These rules may restrict some values of RDF triples or limit the number of triples allowed for identical subjects.

3.6.2 Integrity Violation Example

Consider an RDF dataset, based on the LUBM [125] benchmark, containing students, departments, professors, and universities, where each student is enrolled in one university and has one professor as an advisor. I show in Figure 3.14 an example of this dataset, where John, Sally, and Paul are students of either Yale or UCLA, while Professor William is their advisor. Let me assume an integrity constraint that states that there cannot exist two graduate students in two different universities and have

Figure 3.14: An example RDF graph
the same professor as an advisor. According to this rule, there are two violations in this RDF dataset: (Paul, John) and (Paul, Sally). The query graph of this rule is shown in Figure 3.15 and its SPARQL translation, the query language for RDF data, is as following:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/zhp2/2004/0401/univbench.owl#>

SELECT ?stud1 ?stud2 WHERE {
  ?stud1 rdf:type ub:GraduateStudent . ?stud2 rdf:type ub:GraduateStudent .
  ?dept1 rdf:type ub:Department . ?dept2 rdf:type ub:Department .
}
```

### 3.6.3 Violation Detection Plan in BigDansing

Traditional declarative rules, i.e., FD and DC, cannot represent the above integrity constraint, shown in Figure 3.15 for RDF data. Instead, it may be implemented in BigDansing by using a set of 9 UDFs: 1 Scope operator, 3 Block operators, 3 Iterate operators, 1 Detect operator, and 1 GenFix operator. I show in Figure 3.16 the generated logical plan for the UDFs, while I show in Figure 3.17 the role of each UDF. The plan starts with the Scope operator which removes redundant RDF triples. That is, it only passes to the next operator triples with advisor, memberOf and subOrgOf in their predicate value, or of type Graduate Student and Department. After that, I apply the first Block to group triples based on the subject value (i.e., John, Sally, Paul, U/CS and Y/CS) followed by the first Iterate operator to join the triples of each group. In this step, as shown in Figure 3.17, triples of type Graduate Student gets
the advisor name (as item 2) and the department name (as item 3), while universities are joined with the department name (i.e., UCLA is joined with U/CS). Results are then passed to the second Block and Iterate operators to group incoming triples based on the department name (item 3), where students get the name of their university (i.e., John gets UCLA). After that, the third Block and Iterate operators are applied on the output to group students by the name of their advisor (i.e., William). This allows the Detect operator to generate violations based on students having the same advisor but study in different universities. Finally, the GenFix operator suggests fixing the University names in incoming violations.
Figure 3.17: Data transformations within the logical operators in Figure 3.16
3.6.4 Experiments

I generate an LUBM-based dirty RDF data that partially violates the integrity constraint defined in Figure 3.15. To achieve this, I shuffle the advisor of 10% of the Graduate Student triples on the RDF datasets generated by LUBM. I execute in BigDansing the logical plan of the integrity constraint, described in Figure 3.16, on different sizes of the dirty dataset and compare BigDansing’s performance to S2RDF [122], a state-of-the-art Spark-based RDF engine. The results of this experiment are shown in Figure 3.18, where I show the pre-processing time and the violation detection time for both systems.

BigDansing does not require any data pre-processing. S2RDF, on the other hand, requires RDF partitioning to decompose the input into smaller tables to minimize the overhead of query evaluation. By comparing the total runtime of both systems, I find that BigDansing is at least an order of magnitude faster than the total time of S2RDF. In other words, BigDansing may run ten iterations of viola-
tion detection on the input RDF before S2RDF runs a single query. When excluding the pre-processing time of S2RDF, BigDansing becomes only 1.5 times slower than S2RDF. The results of this experiment show that BigDansing is able to handle schema-less datasets as efficient as a specialized state-of-the-art system that runs on the same general-purpose distributed data processing system.
Chapter 4

Fast and Scalable Inequality Joins

In this chapter, I present IEJoin as novel, fast and space efficient inequality join algorithm to handle sophisticated error discovery in BigDANSING. First, I explain how to achieve fast inequality joins through sorted arrays, bit-arrays and permutation arrays (Chapter 4.1). After that, I illustrate the centralized version of IEJoin and discuss related optimization techniques to significantly speedup the computation (Chapter 4.2). I also propose a selectivity estimation algorithm for IEJoin to improve the performance of multi-predicate queries and multi-way joins (Chapter 4.3). Moreover, I introduce an incremental version of IEJoin to handle fast data updates on the join result assuming streaming or dynamic applications (Chapter 4.4). To perform IEJoin within BigDANSING, I develop a scalable implementation of IEJoin that is compatible with state-of-the-art general-purpose distributed data processing systems (Chapter 4.5). Finally, I implement IEJoin on PostgreSQL, Spark SQL, BigDANSING, and Rheem (Chapter 4.6), and then conduct extensive experiments to study the performance of my algorithm against well-known optimization techniques. The results in this chapter show that my proposed solution is more general, scalable, and orders of magnitude faster than the state-of-the-art solutions (Chapter 4.7).

4.1 Solution Overview

In this section, the discussion is restricted to queries with inequality predicates only. Each predicate is of the form: \( A_i \ op \ B_i \) where \( A_i \) (resp., \( B_i \)) is an attribute in a
relation $R$ (resp., $S$) and $\text{op}$ is an inequality operator in the set \{<, >, \leq, \geq\}.

**Example 3**: [Single predicate] Consider the west table in Figure 1.2 and an inequality self-join query $Q_s$:

$$Q_s : \text{SELECT } s_1.t\_id, s_2.t\_id \text{ FROM west } s_1, \text{ west } s_2$$

$$\text{WHERE } s_1.\text{time} > s_2.\text{time};$$

Query $Q_s$ returns a set of pairs $\{(s_i, s_j)\}$ where $s_i$ takes more time than $s_j$; the result is $\{(s_1, s_3), (s_1, s_4), (s_2, s_1), (s_2, s_3), (s_2, s_4), (s_4, s_3)\}$. □

A natural idea to handle inequality join on one attribute is to leverage a sorted array. For instance, west tuples are sorted on time in ascending order in array $L_1: (s_3, s_4, s_1, s_2)$. $L[i]$ is denoted as the $i$-th element in array $L$, and $L[i, j]$ as its sub-array from position $i$ to position $j$. Given a tuple $s$, any tuple at $L_1[k]$ ($k \in [1, i - 1]$) has time value less than $L_1[i]$, the position of $s$ in $L_1$. Consider Example 3 tuple $s_1$ in position $L_1[3]$ joins with tuples in positions $L_1[1, 2]$, namely $s_3$ and $s_4$.

**Example 4**: [Two predicates] Let me now consider a self-join with two inequality conditions:

$$Q_p : \text{SELECT } s_1.t\_id, s_2.t\_id \text{ FROM west } s_1, \text{ west } s_2$$

$$\text{WHERE } s_1.\text{time} > s_2.\text{time} \text{ AND } s_1.\text{cost} < s_2.\text{cost};$$

$Q_p$ returns pairs $(s_i, s_j)$ where $s_i$ takes more time but pays less than $s_j$; the result is $\{(s_1, s_3), (s_4, s_3)\}$. □

Similar to attribute time in Example 3, attribute cost is sorted in ascending order into array $L_2: (s_4, s_1, s_3, s_2)$, as shown below.

$$L_1 \begin{bmatrix} s_3(80) & s_4(90) & s_1(100) & s_2(140) \end{bmatrix} \text{ (sort ↑ on time)}$$

$$L_2 \begin{bmatrix} s_4(5) & s_1(6) & s_3(10) & s_2(11) \end{bmatrix} \text{ (sort ↑ on cost)}$$
Thus, given a tuple $s$ whose position in $L_2$ is $j$, any tuple $L_2[l]$ ($l \in [j+1, n]$) has a higher cost than $s$, where $n$ is the size of the input relation. My finding here is as follows. For any tuple $s'$, to form a join result $(s, s')$ with tuple $s$, the following two conditions must be satisfied: (i) $s'$ is on the left of $s$ in $L_1$, i.e., $s$ has a larger value for time than $s'$, and (ii) $s'$ is on the right of $s$ in $L_2$, i.e., $s$ has a smaller value for cost than $s'$. Thus, all tuples in the intersection of $L_1[1, i-1]$ and $L_2[j+1, n]$ satisfy these two conditions and belong to the join result. For example, $s_4$’s position in $L_1$ (resp. $L_2$) is 2 (resp. 1). Hence, $L_1[1, 2-1] = \langle s_3 \rangle$ and $L_2[1+1, 4] = \langle s_1, s_3, s_2 \rangle$, and their intersection is $\{s_3\}$, producing $(s_4, s_3)$. To obtain the final result, I just have to repeat the above process for each tuple.

The challenge is how to perform the intersection operation, mentioned above, in a more efficient manner. There already exist several indices, such as $R$-tree and $B^+$-tree, that might help. $R$-tree is ideal for supporting two or higher dimensional range queries. However, the non-clustered nature of $R$-trees makes them inadequate for inequality joins; it is impossible to avoid random I/O access when retrieving join results. $B^+$-tree is a clustered index. The advantage is that for each tuple, only a sequential disk scan is required to retrieve relevant tuples. However, this is needed to be repeated $n$ times, where $n$ is the number of tuples, which is prohibitively expensive. When confronted with such problems, one common practice is to use space-efficient and CPU-friendly indices; in this chapter, I make use of bit-arrays.

As discussed earlier, the idea of handling an inequality join on one attribute is to leverage a sorted array, as shown in Example 3. When two different attributes appear in the join, to leverage a similar idea, a natural solution is to use a permutation array between two sorted arrays $L_1$ and $L_2$. Given the $i$-th element in $L_1$, a permutation array can tell its corresponding position in $L_2$ in constant time. Moreover, when visiting items in $L_1$, it is important to keep track of the items seen so far, for which I use a bit-array to make such a connection.
(1) Initialization

<table>
<thead>
<tr>
<th>$L_1$</th>
<th>$s_3(80)$</th>
<th>$s_4(90)$</th>
<th>$s_1(100)$</th>
<th>$s_2(140)$</th>
<th>(sort ↑ on time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_2$</td>
<td>$s_4(5)$</td>
<td>$s_1(6)$</td>
<td>$s_3(10)$</td>
<td>$s_2(11)$</td>
<td>(sort ↑ on cost)</td>
</tr>
</tbody>
</table>

$P = [2, 3, 1, 4]$ (permutation array) $B = [0, 0, 0, 0]$ (bit-array)

(2) Visit tuples w.r.t. $L_2$

(a) $\cdot \rightarrow$ $B = [0, 0, 0, 0]$ $\Rightarrow$ $[0, 1, 0, 0]$ Output:

(b) $\cdot \rightarrow$ $B = [0, 1, 0, 0]$ $\Rightarrow$ $[0, 1, 1, 0]$ Output:

(c) $\cdot \rightarrow$ $B = [0, 1, 1, 0]$ $\Rightarrow$ $[1, 1, 1, 0]$ Output: $(s_4, s_3), (s_1, s_3)$

(d) $\cdot$ $B = [1, 1, 1, 0]$ $\Rightarrow$ $[1, 1, 1, 1]$ Output:

Figure 4.1: IEJoin process for query $Q_p$

Generally speaking, the proposed inequality join sorts relation west on time and cost, creates a permutation array for cost w.r.t. time and leverages a bit-array to produce join results. The algorithm is briefly presented below, while detailed discussions are deferred to Chapter 4.2. Figure 4.1 describes the inequality join process.

S1. [Initialization] Sort both time and cost values in ascending order, as depicted by $L_1$ and $L_2$, respectively. While sorting, compute a permutation (reordering) array of elements of $L_2$ in $L_1$, as shown by $P$. For example, the first element of $L_2$ (i.e., $s_4$) corresponds to position 2 in $L_1$. Hence, $P[1] = 2$. Initialize a bit-array $B$ with length $n$ and set all bits to 0, as shown by $B$ with array indices reported above the cells and corresponding tuples reported below them.

S2. [Visit tuples in the order of $L_2$] Scan the permutation array $P$ and operate on the bit-array.
(a) **Visit P[1]**. First visit tuple $s_4$ (1$^{st}$ element in $L_2$) and check in $P$ what is the position of $s_4$ in $L_1$ (*i.e.*, position 2). Then go to $B[2]$ and scan all bits in positions higher than 2. As all $B[i] = 0$ for $i > 2$, there is no tuple that satisfies the join condition of $Q_p$ *w.r.t.* $s_4$. Finish this visit by setting $B[2] = 1$, which indicates that tuple $s_4$ has been visited.

(b) **Visit P[2]**. This is for tuple $s_1$. It processes $s_1$ in a similar manner as $s_4$, without emitting any result.

(c) **Visit P[3]**. This visit corresponds to tuple $s_3$. Each non-zero bit on the right of $s_3$ (highlighted by grey cells) corresponds to a join result, because each marked cell corresponds to a tuple that pays less cost (*i.e.*, being visited first) but takes more time (*i.e.*, on the right side of its position). It thus outputs $(s_4, s_3)$ and $(s_1, s_3)$.

(d) **Visit P[4]**. This visit corresponds to tuple $s_2$ and does not return any result.

The final result of $Q_p$ is the union of all the intermediate results from the above steps, *i.e.*, \{(s_4, s_3), (s_1, s_3)\}.

A few observations make the discussed solution appealing. First, there are many efficient techniques for sorting large arrays, *e.g.*, GPUTeraSort [127]. In addition, after getting the permutation array, I only need to sequentially scan it once. Hence, the permutation array can be stored on disk, in case there is not enough memory. Only the bit-array is required to stay in memory, to avoid random disk I/Os. Thus, to execute queries $Q_s$ and $Q_p$ on 1 billion tuples, theoretically, the bit-array only needs 1 billion bits (*i.e.*, 125 MB) of memory.

### 4.2 Centralized Algorithms

In this section, I describe IEJoin, which is the implementation of the inequality join algorithm using permutation arrays and bit-arrays. I start the discussion by present-
Algorithm 3: IEJoin

\[\textbf{Algorithm 3: IEJoin}\]

\textbf{input} : query \(Q\) with 2 join predicates \(t_1.X \ \text{op}_1 \ t_2.X'\) and \(t_1.Y \ \text{op}_2 \ t_2.Y'\), tables \(T, T'\) of sizes \(m\) and \(n\) resp.

\textbf{output}: a list of tuple pairs \((t_i, t_j)\)

1. let \(L_1\) (resp. \(L_2\)) be the array of \(X\) (resp. \(Y\)) in \(T\)
2. let \(L'_1\) (resp. \(L'_2\)) be the array of \(X'\) (resp. \(Y'\)) in \(T'\)
3. if \((\text{op}_1 \in \{>, \geq\})\) sort \(L_1, L'_1\) in descending order
4. else if \((\text{op}_1 \in \{<, \leq\})\) sort \(L_1, L'_1\) in ascending order
5. if \((\text{op}_2 \in \{>, \geq\})\) sort \(L_2, L'_2\) in ascending order
6. else if \((\text{op}_2 \in \{<, \leq\})\) sort \(L_2, L'_2\) in descending order
7. compute the permutation array \(P\) of \(L_2\) w.r.t. \(L_1\)
8. compute the permutation array \(P'\) of \(L'_2\) w.r.t. \(L'_1\)
9. compute the offset array \(O_1\) of \(L_1\) w.r.t. \(L'_1\)
10. compute the offset array \(O_2\) of \(L_2\) w.r.t. \(L'_2\)
11. initialize bit-array \(B'\) (\(|B'| = n\)), and set all bits to 0
12. initialize \(\text{join_result}\) as an empty list for tuple pairs
13. if \((\text{op}_1 \in \{\leq, \geq\} \ \text{and} \ \text{op}_2 \in \{\leq, \geq\})\) eqOff = 0
14. else eqOff = 1
15. for \((i \leftarrow 1 \ \text{to} \ m)\) do
16. \hspace{0.5cm} off_2 \leftarrow O_2[i]
17. \hspace{0.5cm} for \(j \leftarrow 1 \ \text{to} \ \min(\text{off}_2, \text{size}(L_2))\) do
18. \hspace{1cm} \(B'[P'[j]] \leftarrow 1\)
19. \hspace{0.5cm} off_1 \leftarrow O_1[P[i]]
20. \hspace{0.5cm} for \(k \leftarrow \text{off}_1 + \text{eqOff} \ \text{to} \ n\) do
21. \hspace{1cm} if \(B'[k] = 1\) then
22. \hspace{1.5cm} add tuples w.r.t. \((L_2[i], L'_2[k])\) to \(\text{join_result}\)
23. \hspace{0.5cm} end
24. \hspace{0.5cm} end
25. \hspace{0.5cm} end
26. \hspace{0.5cm} end
27. \hspace{0.5cm} end
28. \hspace{0.5cm} end
29. \hspace{0.5cm} end
30. return \(\text{join_result}\)

I describe in this section the standard form of IEJoin, which joins two relations using two inequality predicates.

\textbf{Algorithm}. The algorithm, IEJoin, is shown in Algorithm 3. It takes a query \(Q\) with

The case of two-way joins with two operators from \(\{<, >, \leq, \geq\}\), followed by the case of using not equal operator \((i.e., \neq)\) and outer joins (Chapter 4.2.1). After that, I discuss the special case of self-joins (Chapter 4.2.2), and conclude by describing optimization techniques to the algorithm’s data structures (Chapter 4.2.3).

4.2.1 IEJoin

I describe in this section the standard form of IEJoin, which joins two relations using two inequality predicates.
two inequality join conditions as input and returns a set of result pairs. It first sorts the attribute values to be joined (lines 3-6), computes the permutation array (lines 7-8) and two offset arrays (lines 9-10). The details of computing the permutation arrays are deferred to Chapter 4.6. Each element of an offset records the relative position from \( L_1 \) (resp. \( L_2 \)) in \( L'_1 \) (resp. \( L'_2 \)). The offset array is computed by a linear scan of both sorted arrays (e.g., \( L_1 \) and \( L'_1 \)). The algorithm also sets up the bit-array (line 11) as well as the result set (line 12). In addition, it sets an offset variable to distinguish between the inequality operators with or without equality conditions (lines 13-14).

It then visits the values in \( L_2 \) in the appropriate order, which is to sequentially scan the permutation array from left to right (lines 15-22). For each tuple visited in \( L_2 \), it first sets all bits for those \( t \) in \( T' \) whose \( Y' \) values are smaller than the \( Y \) value of the current tuple in \( T \) (lines 16-18), i.e., those tuples in \( T' \) that satisfy the second
join condition. It then uses the other offset array to find those tuples in $T'$ that also satisfy the first join condition (lines 19-22). It finally returns all join results (line 23).

$Q_t:$ \texttt{SELECT east.id, west.t_id FROM east, west WHERE east.dur < west.time AND east.rev > west.cost;}

Example 5: Figure 4.2 shows how Algorithm 3 processes $Q_t$ from above. In initialization (step 1), $L_1$, $L_2$, $L'_1$ and $L'_2$ are sorted; $P$ (resp. $P'$) is the permutation array between $L_1$ and $L_2$ (resp. $L'_1$ and $L'_2$), where the details of computation are given in implementation details for PostgreSQL in Chapter 4.6.1 Table 4.2; $O_1$ (resp. $O_2$) is the offset array of $L_1$ relative to $L'_1$ (resp. $L_2$ relative to $L'_2$), e.g., $O_1[1] = 2$ means that the relative position of value $L_1[1] = 90$ in $L'_1$ is 2. Clearly, $O_1$ (resp. $O_2$) can be computed by sequentially scanning $L_1$ and $L_1$ (resp. $L_2$ and $L'_2$); and $B'$ is the bit-array with all bits initialized to be 0.

After the initialization, the algorithm starts visiting tuples \textit{w.r.t.} $L_2$ (step(2)). For example, when visiting the first item in $L_2$ ($r_3$) in step (2)(a), it first finds its relative position in $L'_2$ at step (2)(a)(i). Then it visits all tuples in $L'_2$ whose cost values are no larger than $r_3[\text{rev}]$ at step (2)(a)(ii). Afterwards, it uses the relative position of $r_3[\text{dur}]$ at $L'_1$ (step (2)(a)(iii)) to populate all join results (step (2)(a)(iv)). The same process applies to $r_1$ (step (2)(b)) and $r_2$ (step (2)(c)), and the only result is returned at step (2)(c)(v).

\begin{flushright}
$\Box$
\end{flushright}

Correctness. The algorithm terminates and the results satisfy the join condition. For any tuple pair ($r_i, s_j$) that should be a result, $s_j$ will be visited first and its corresponding bit is set to 1 (lines 17-18). Afterwards, $r_i$ will be visited and the result ($r_i, s_j$) will be identified (lines 20-22) by the algorithm.

Complexity. Sorting arrays and computing their permutation array together are in $O(m \cdot \log m + n \cdot \log n)$ time, where $m$ and $n$ are the sizes of the two input relations.
(lines 3-8). Computing the offset arrays will take linear time using sort-merge (lines 9-10). The outer loop will take $O(m \cdot n)$ time (lines 15-22). Hence, the total time complexity of the algorithm is $O(m \cdot \log m + n \cdot \log n + m \cdot n)$. It is straightforward to see that the total space complexity is $O(m + n)$.

**Not equal operator.** In the case of the not equal “$\neq$” operator, I simply rewrite the query as two queries, with the $(>)$ and $(<)$ operators, respectively. I then merge the results of these two queries through the `UNION ALL` SQL command.

*Example 6:* Consider the following query with one $\neq$ condition:

$$Q_k : \text{SELECT } r.\text{id}, s.\text{id} \text{ FROM Events } r, \text{Events } s$$
$$\text{WHERE } r.\text{start} \leq s.\text{end} \text{ AND } r.\text{end} \neq s.\text{start};$$

I translate $Q_k$ into the union of two queries as follows:

$$Q'_k : \text{SELECT } r.\text{id}, s.\text{id} \text{ FROM Events } r, \text{Events } s$$
$$\text{WHERE } r.\text{start} \leq s.\text{end} \text{ AND } r.\text{end} < s.\text{start}$$
$$\text{UNION ALL}$$
$$\text{SELECT } r.\text{id}, s.\text{id} \text{ FROM Events } r, \text{Events } s$$
$$\text{WHERE } r.\text{start} \leq s.\text{end} \text{ AND } r.\text{end} > s.\text{start};$$

The performance of IEJoin for queries with a “$\neq$” operator strongly depends on the attribute domain itself. Attributes with smaller ranges will typically lead to higher selectivity, while larger ranges to lower selectivity. Note that as higher selective attributes produce fewer results, compared to lower ones, they have a faster IEJoin runtime.

**Outer joins.** IEJoin can also support left, right, and full outer joins. For left outer joins, any tuple that does not find any matches while scanning the bit-array (lines 20-22 in Algorithm 3) is paired with a Null value. The right outer join is processed
Algorithm 4: IESelfJoin

**input**: query $Q$ with 2 join predicates $t_1.X \ op_1 t_2.X$ and $t_1.Y \ op_2 t_2.Y$, table $T$ of size $n$

**output**: a list of tuple pairs $(t_i, t_j)$

1. let $L_1$ (resp. $L_2$) be the array of column $X$ (resp. $Y$)
2. if $(\text{op}_1 \in \{>, \geq\})$ sort $L_1$ in ascending order
3. else if $(\text{op}_1 \in \{<, \leq\})$ sort $L_1$ in descending order
4. if $(\text{op}_2 \in \{>, \geq\})$ sort $L_2$ in descending order
5. else if $(\text{op}_2 \in \{<, \leq\})$ sort $L_2$ in ascending order
6. compute the permutation array $P$ of $L_2$ w.r.t. $L_1$
7. initialize bit-array $B$ ($|B| = n$), and set all bits to 0
8. initialize join_result as an empty list for tuple pairs
9. if $(\text{op}_1 \in \{\leq, \geq\} \text{ and } \text{op}_2 \in \{\leq, \geq\})$ eqOff = 0
10. else eqOff = 1
11. for $(i \leftarrow 1 \text{ to } n)$ do
12.   pos $\leftarrow P[i]$
13.   $B[$pos$] \leftarrow 1$
14.   for $(j \leftarrow \text{pos} + \text{eqOff} \text{ to } n)$ do
15.     if $B[j] = 1$ then
16.       add tuples w.r.t. $(L_1[j], L_1[P[i]])$ to join_result
17. return join_result

by flipping the order of input relations, executing a normal left outer join and then reversing the order of the join result. The IEJoin output should return $(L_2'[k], L_2[i])$ or $(NULL, L_2[i])$ instead of the original order (line 22 in Algorithm 3) to generate the correct result for the right outer join. The full outer join is translated into one left outer join that includes the normal IEJoin output, and one right outer join that only emits results with null values.

### 4.2.2 IESelfJoin

In this section, I present the algorithm for self-join queries with two inequality operators.

**Algorithm**. IESelfJoin (Algorithm 4) takes a self-join inequality query $Q$ and returns a set of result pairs. The algorithm first sorts the two lists of attributes to be joined
(lines 2-5), computes the permutation array (line 6), and sets up the bit-array (line 7) as well as the result set (line 8). It also sets an offset variable to distinguish inequality operators with or without equality (lines 9-10). It then visits the values in \( L_2 \) in the desired order, \textit{i.e.}, sequentially scan the permutation array from left to right (lines 11-16). For each tuple visited in \( L_2 \), it needs to find all tuples whose \( X \) values satisfy the join condition. This is performed by first locating its corresponding position in \( L_1 \) via looking up the permutation array (line 12) and marked in the bit-array (line 13). Since the bit-array and \( L_1 \) have a one-to-one positional correspondence, the tuples on the right of \( \text{pos} \) will satisfy the join condition on \( X \) (lines 14-16), and these tuples will also satisfy the join condition on \( Y \) if they have been visited before (line 15). Such tuples will be joined with the tuple currently being visited as results (line 16). It finally returns all join results (line 17).

Note that the different sorting orders, \textit{i.e.}, ascending or descending for attribute \( X \) and \( Y \) in lines 2-5, are chosen to satisfy various inequality operators. One may observe that if the database contains duplicated values, when sorting one attribute \( X \), its corresponding value in attribute \( Y \) should be considered, and vice versa, in order to preserve both orders for correct join result. Hence, in IESelfJoin, when sorting \( X \), I use an algorithm that also takes \( Y \) as the secondary key. Specifically, when some \( X \) values are equal, their sorting orders are decided by their \( Y \) values (lines 2-3), similarly for the other way around (lines 4-5). Note that if both attributes are duplicates, I make sure that the sorting of the attributes is consistent by relying on the internal record (tuple) ids. In particular, when the values are equal both in \( X \) and \( Y \), I sort them in increasing order according to their internal id.

Moreover, I show in Table \ref{tab:sorting-orders} the sorting orders for \( \text{op}_1 \)'s secondary key (\( Y \)) and \( \text{op}_2 \)'s secondary key (\( X \)) when the dataset contains duplicate values. For example, if \( \text{op}_1 \)'s condition is \((\leq)\) and \( \text{op}_2 \)'s condition is \((>)\), according to Table \ref{tab:sorting-orders} equal values in \( \text{op}_1 \) are sorted based on ascending order of their \( Y \) values while \( \text{op}_2 \)'s equal
Table 4.1: Secondary key sorting order for $Y/X$ in $op_1/op_2$

<table>
<thead>
<tr>
<th>$op_1$ sort order</th>
<th>&lt;</th>
<th>&gt;</th>
<th>$\leq$</th>
<th>$\geq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Asc/Des</td>
<td>Des/Des</td>
<td>Asc/Asc</td>
<td>Des/Asc</td>
</tr>
<tr>
<td>&gt;</td>
<td>Asc/Asc</td>
<td>Des/Asc</td>
<td>Asc/Des</td>
<td>Des/Des</td>
</tr>
<tr>
<td>$\leq$</td>
<td>Des/Des</td>
<td>Asc/Asc</td>
<td>Des/Asc</td>
<td>Asc/Asc</td>
</tr>
<tr>
<td>$\geq$</td>
<td>Des/Asc</td>
<td>Asc/Asc</td>
<td>Des/Asc</td>
<td>Asc/Des</td>
</tr>
</tbody>
</table>

values are sorted on descending order of their $X$ values. Please refer to the example in Chapter 4.1 for query $Q_p$ using IESelfJoin.

**Correctness.** It is easy to check that the algorithm will terminate and each result in join_result satisfies the join condition. For completeness, observe the following. For any tuple pair $(t_1, t_2)$ that should be in the result, $t_2$ is visited first and its corresponding bit is set to 1 (line 13). Afterwards, $t_1$ is visited and the result $(t_1, t_2)$ is identified (lines 15-16) by IESelfJoin.

**Complexity.** Sorting two arrays and computing their permutation array is in $O(n \cdot \log n)$ time (lines 2-8). Scanning the permutation array and scanning the bit-array for each visited tuple run in $O(n^2)$ time (lines 11-16). Hence, in total, the time complexity of IESelfJoin is $O(n^2)$. It is easy to see that the space complexity of IESelfJoin is $O(n)$.

### 4.2.3 Enhancements

I discuss two techniques to improve performance: (i) Indices to improve the lookup performance for the bit-array. (ii) Union arrays to improve data locality and reduce the data to be loaded into the cache.

**Bitmap index to improve bit-array scan.** An analysis on both IEJoin and IESelfJoin shows that for each visited value (*i.e.*, lines 20-22 in Algorithm 3 and lines 14-16 in Algorithm 4), I need to scan all the bits on the right of the current position. When the query selectivity is low, this is unavoidable for producing the
correct results. However, when the query selectivity is high, iteratively scanning a long sequence of 0’s will be a performance bottleneck. I thus adopt a bitmap to guide which parts of the bit-array should be visited.

Given a bit-array $B$ of size $n$ and a predefined chunk size $c$, my bitmap is a bit-array with size $\lceil n/c \rceil$ where each bit corresponds to a chunk in $B$, with 1 indicating that the chunk contains at least a 1, and 0 otherwise.

**Example 7:** Consider the bit-array $B$ in Figure 4.3. Assume that the chunk size $c = 4$. The bit-array $B$ will be partitioned into four chunks $C_1$–$C_4$. Its bitmap is shown above $B$ in the figure and consists of 4 bits. I consider two cases.

**Case 1:** visit $B[6]$, in which case I need to find all the 1’s in $B[i]$ for $i > 6$. The bitmap tells that only chunk 2 needs to be checked, and it is safe to ignore chunks 3 and 4.

**Case 2:** visit $B[9]$, the bitmap tells that there is no need to scan $B$, since there cannot be any $B[j]$ where $B[j] = 1$ and $j > 9$.

To further improve the bitmap, I avoid unnecessary scanning by stopping at the maximum modified value in the bitmap index. For example, in Figure 4.3, I maintain a max scan index variable ($MaxIndex$) with value 2 to stop the bitmap from unnecessarily scanning bitmap indices 3 and 4. The initial value of this variable is 0, which means that both the bitmap and the bit-array are empty. As I iteratively modify the
bit-array (line 18 in Algorithm 3 and line 13 in Algorithm 4) and edit the bitmap, I make sure to update the max scan index variable if the current updated filter index is larger than the last recorded max filter index.

**Union arrays on join attributes.** In testing Algorithm 3, I found that there are several cache loads and stores. A deeper analysis shows that the extra cache loads and stores may be caused by cache misses when sequentially visiting different arrays. Take Figure 4.2 for example. In step (2)(a), I visit arrays $L_2$, $O_2$, $P'$, $P$ and $O_1$ in sequence, with each causing at least one cache miss. Step (2)(b) and step (2)(c) show a similar behavior. An intuitive solution is to merge the arrays on join attributes and sort them together. Again, consider Figure 4.2. I can merge $L_1$ and $L'_1$ into one array and sort them, which will result in $\langle s_3(80), r_3(90), s_4(90), r_2(100), s_1(100), r_1(140), s_2(140) \rangle$. Similarly, I can merge $L_2$ and $L'_2$, and $P$ and $P'$. Also, $O_1$ and $O_2$ are not needed in this case, and $B'$ needs to be extended to be aligned with the merged arrays. This solution is similar to IESelfJoin (Chapter 4.2.2). However, I need to prune join results for tuples that come from the same table. This can be easily done using a Boolean flag for each position, where 0 (resp. 1) denotes that the corresponding value is from the first (resp. the second) table. My experiments (Chapter 4.7.5) show that this simple union can significantly reduce the number of cache misses, and thus improve execution time.

### 4.3 Query Optimization

I introduce an approach to estimate the selectivity of inequality join predicates that is then used to optimize inequality join queries. In particular, I tackle the cases of selecting the best join predicate when joining two relations on more than two join predicates and selecting the best join order for a multi-way IEJoin. Note that my goal is not to build a full-fledged query optimizer and implement it in an existing system, but rather to propose a simple, yet efficient, approach for optimizing joins.
Algorithm 5: IEJoin Selectivity Estimation

**input**: join predicates \( t_1.X \, \text{op}_1 \, t_2.X' \) and \( t_1.Y \, \text{op}_2 \, t_2.Y' \), sample tables \( T_s \& T_{s'} \), block size \( n \)

**output**: Number of overlapping blocks

1. `overlappingBlocks ← 0`
2. **if** \( \{\text{op}_1 \in \{<, \leq\}\} \) **then**
3. sort \( T_s \& T_{s'} \) in descending order on \( X \) and \( X' \)
4. **else if** \( \{\text{op}_1 \in \{>, \geq\}\} \) **then**
5. sort \( T_s \& T_{s'} \) in ascending order on \( X \) and \( X' \)
6. Partition \( T_s \& T_{s'} \) into \( B \& B' \) blocks of size \( n \)
7. **for** \( (i \leftarrow 1 \, \text{to} \, B) \) **do**
8. **for** \( (j \leftarrow 1 \, \text{to} \, B') \) **do**
9. if \( T_{s_i} \cap T_{s'_j} \) **then**
10. increment `overlappingBlocks` by 1
11. **return** `overlappingBlocks`

with inequality conditions.

### 4.3.1 Selectivity Estimation

To estimate the selectivity of an inequality join predicate, I simply count the number of overlapping sorted blocks obtained from a sample of the input tables. I assume that the join has two inequality predicates as in Algorithm 3. I first obtain a sample from the two tables to join, using uniform random sampling\(^1\). I give the samples as input to Algorithm 5. I then sort the two samples based on the attributes involved in one of the join predicates (lines 2-5), while using the remaining join attributes to sort duplicate tuples. Next, I divide each sample into smaller blocks (line 6) and test all combinations of block pairs for potential overlaps based on min/max values of all of the join attributes in each block (lines 7-10). The idea is that overlapping blocks may generate results for a given query while non-overlapping blocks cannot generate results at all. Finally, I return the number of overlapping blocks as the selectivity.

\(^1\)I experimentally show in Chapter 4.7.7 that my algorithm requires only 1% of the input data to be accurate.
estimation of the inequality join (line 11). The lower the number of overlapping blocks, the higher the selectivity of the inequality join condition.

While I cannot determine the exact size of the join output in overlapping blocks before execution, sorting helps with good estimates. This is because sorting blocks naturally increases the locality of input relations as I consider inequality join conditions. Thus, the number of overlapping blocks is a good approximation of the selectivity of an inequality join condition.

4.3.2 Join Optimization with Multiple Predicates

Given an inequality join query on two relations and with more than two join predicates, I need to determine which two predicates should be joined first to minimize the cost. The remaining predicates will be evaluated on the result of the join. To this end, I need to estimate the selectivity of all pair combinations of the join predicates in the query. For example, I compare the selectivity estimation of three pair combinations ($\Join_1 \Join_3$) for $Q_w$,

$Q_w$: SELECT $s_1.t_id, s_2.t_id$ FROM west $s_1$, west $s_2$

WHERE $s_1.time > s_2.time$ and $s_1.cost < s_2.cost$

and $s_1.totalUsers < s_2.totalUsers$;

as follows:

$\Join_1$: $s_1.time > s_2.time$ \& \& $s_1.cost < s_2.cost$

$\Join_2$: $s_1.time > s_2.time$ \& \& $s_1.totalUsers < s_2.totalUsers$

$\Join_3$: $s_1.cost < s_2.cost$ \& \& $s_1.totalUsers < s_2.totalUsers$

I need to test $\binom{N}{2}$ combinations to choose the best join predicates for a given query with $N$ input join predicates. Clearly, a large $N$ would increase the overhead
of the selectivity estimation. To lower this overhead, I reuse the sorted input relations in multiple instances of Algorithm 5 that share a similar join predicate. For example, IEJoin can sort on attribute \texttt{time} (or \texttt{totalUsers}) to reuse it in join predicates \langle \texttt{time}, \texttt{totalUsers} \rangle and \langle \texttt{time}, \texttt{cost} \rangle (resp. \langle \texttt{time}, \texttt{totalUsers} \rangle and \langle \texttt{cost}, \texttt{totalUsers} \rangle) as they both share attribute \texttt{time} (resp. \texttt{totalUsers}). Note that sorting based on one attribute or another does not impact the selectivity estimation of the pair combination.

### 4.3.3 Multi-way Join Optimization

For multi-way inequality joins, I follow a common approach in optimizing such joins, such as in [82]. Generally speaking, I execute a multi-way inequality join as a series of two-way inequality joins formed as a left-deep plan. I adopt a greedy approach where I choose the order of the two-way joins based on their estimated selectivity, i.e., the two-way joins with higher selectivity (lowest number of overlapping blocks as computed by Algorithm 5) are pushed down in the plan.

Algorithm 6 builds the execution plan for a multi-way join query. I first estimate the selectivity of all possible combinations of two-way joins using Algorithm 5 (lines 3-6). I discard Cartesian products. At level one of the left-deep plan, I pick the two relations with the most selective inequality join (lines 8-11). I then proceed by selecting the one relation, among the remaining ones, that would deliver the most selective inequality join if joined with the previous level in the plan (lines 12-18). This is performed by selecting the relation that has the smallest number of overlapped blocks when joined with a relation from the previous level. I only consider joins that share an inequality predicate with one relation in the previous levels (lines 14-16). In each level, I append the most selective two-way join to the final execution plan (lines 17-18) and remove it from the pool of available joins (line 19). By repeating this process until there are no more relations to join, I compute a full IEJoin order.
Algorithm 6 returns an array that describes the left-deep plan. The plan is obtained by joining the relations in the order they appear in the array, i.e., first join \( \text{plan}[1] \) with \( \text{plan}[2] \), then the result with \( \text{plan}[3] \) and so on.

```
Algorithm 6: Multi-way IEJoin planner

input : input relations \( R \)
output: join plan

1. \( n \leftarrow |R| \)
2. \( \text{Est} \leftarrow \text{empty set} \)
3. for \( (i \leftarrow 1 \text{ to } n) \) do
   4. for \( (j \leftarrow i + 1 \text{ to } n) \) do
      5. if \( R_i \bowtie R_j \) share IEJoin predicates then
         6. \( \text{Est} \leftarrow \text{selectivity estimation of } R_i \bowtie R_j \)
   7. for \( (\text{planIndex} \leftarrow 1 \text{ to } n) \) do
      8. if \( \text{planIndex} = 1 \) then
         9. \( \text{min} \leftarrow \text{smallest estimation} \in \text{Est} \)
        10. \( \text{plan}[1] \leftarrow R_i, \text{ where } i \in \text{min} \)
        11. \( \text{plan}[2] \leftarrow R_j, \text{ where } j \in \text{min} \)
      else if \( \text{planIndex} > 2 \) then
         12. \( \text{Est}' \leftarrow \text{empty set} \)
         13. foreach \( R_i \bowtie R_j \) estimation \( \in \text{Est} \) do
         14. if \( (R_i|R_j) \in \text{plan}[1, \text{planIndex} - 1] \) and \( (R_i \land R_j) \notin \text{plan}[1, \text{planIndex} - 1] \) then
         15. \( \text{Est}' \leftarrow \text{selectivity estimation of } R_i \bowtie R_j \)
         16. \( \text{min} \leftarrow \text{smallest estimation} \in \text{Est}' \)
         17. \( \text{plan}[\text{planIndex}] \leftarrow R_i \text{ (or } R_j), \text{ where } (i \land j) \in \text{min} \text{ and } R_i \text{ (or } R_j) \)
         18. \( \notin \text{plan}[1, \text{planIndex} - 1] \)
        19. remove \( \text{min} \) from \( \text{Est} \)
   20. return \( \text{plan} \)
```

### 4.4 Incremental Inequality Joins

In this section, I present algorithms for incremental inequality joins when the data keeps changing with insertions and deletions. I compute \( Q(D \oplus \Delta D) \), where \( Q \) is an inequality join on \( D \) that contains either one or two relations, and \( \Delta D \) is the data updates that are insertions \( \Delta D^+ \) or deletions \( \Delta D^- \). Adapting my algorithms for
incremental computation faces two challenges: (i) maintaining the sorted arrays; and (ii) only computing results w.r.t. \( \Delta D \).

**Maintain a sorted array.** Given a sorted array \( L \) and an unsorted array \( \Delta L \), the problem is to compute a sorted array for elements in \( L \oplus \Delta L \).

(1) **Sort-merge** \([128]\). The straightforward solution is to first sort \( \Delta L \) and then merge two sorted arrays \( L \) and \( \Delta L \) in linear time. This approach is appropriate for *batch updates*.

(2) **Packed-memory array** \([129]\). A widely adopted approach for maintaining a dynamic set of \( N \) elements in sorted order in \( \Theta(N) \)-sized array is to use Packed-memory array (PMA). The idea is to insert \( \Theta(N) \) empty spaces among the elements of the array such that only a small number of elements need to be shifted around per update. Thus, the number of element moves per update is only \( O(\log_2 N) \). This approach is ideal for continuous query answering in a streaming fashion.

Note that, PMA leaves empty spaces in the array. For instance, the arrays \( L_1 \) and \( L_2 \) in Figure 4.4 are stored using PMAs, where “|”’s denote the empty spaces. Also, PMAs handle deletions \([129]\). Although I use PMAs to maintain the sorted arrays by leaving gaps (*i.e.*, the “|”’s) for future updates, my algorithms are unchanged by simply ignoring these gaps.

Let me start by discussing the incremental inequality join on one relation with insertions. I will then discuss the case with deletions, and then on two relations.

**Incremental IESelfJoin.** I illustrate the idea of computing incremental results on one relation with *insertions* by an example.

*Example 8:* Consider query \( Q_p \) from Example 4 and the data \( T \) used in Figure 4.1. Computing \( Q_p(T) \) is the same as described in Figure 4.1 shown as step (1) in Figure 4.4. Consider a new tuple insertion \( \Delta D^+ = \{s_5(95, 8)\} \). As shown in Fig-
ure 4.4 step (2), it finds the right positions in both sorted arrays, i.e., 95 in position 4 of \(L'_1\), 8 in position 4 of \(L'_2\) and the permutation array is updated.

Similar to the process described in Chapter 4.1, it visits tuples in \(L'_2\) from left to right (Figure 4.4 step (3)).

(a) For all tuples whose time values are less than \(s_5\), that is 8, set all corresponding bits as 1’s since they satisfy one join condition on attribute time.

(b) Visit \(s_5\) and output all results for \((s_5, s_i)\) whose \(s_i\) is on the right of \(s_5\) in \(B\) (i.e., satisfying both join conditions), which is \(\{(s_5, s_1)\}\).

(c) For other tuples on the right of \(s_5\) in \(L'_2\), output a new join result if it contains the new tuple \(s_5\). When visiting \(s_3\), output \(\{(s_3, s_5)\}\).

(d) When visiting \(s_2\), the process is similar to (c), with an empty output.

From the above example, I can see that only new results are produced, i.e., those coming from \(\Delta D^+\). The incremental algorithm using PMA is a simple adaption of Algorithm 4, which is thus omitted here. Moreover, when the update contains a set of insertions, the procedure for each one is the same as described in Example 8.

For deletions \(\Delta D^-\), I use a similar methodology as discussed above to compute updated results. The difference is that, instead of adding these new results, I remove them from old results.

**Incremental IEJoin.** I now discuss how to extend Algorithm 3 on two relations \(R\) and \(S\) to support incremental processing. I will focus on insertions only (\(\Delta R^+\) and \(\Delta S^+\)), since deletions are similar to insertions by only removing results.

Take Figure 4.2 for reference, I perform the following three steps to run IEJoin: (1) maintain the sorted lists w.r.t. the insertions \(\Delta R^+\) and \(\Delta S^+\); (2) maintain the offsets for \(R \oplus \Delta R^+\) relative to \(S \oplus \Delta S^+\); and (3) compute the new results.

Step (1) is the same as discussed in IESelfJoin. Step (2) can be performed in
(1) Compute $Q_p(T)$ (see Figure 4.1)

$L_1 | s_3(80) | s_4(90) | s_1(100) | s_2(140) |

$L_2 | s_4(5) | s_1(6) | s_2(10) | s_2(11) |

$P | 0 | 3 | 5 | 0 | 0 | 1 | 0 | 7$ (permutation array)

(2) Insert $\delta = \{s_5(95, 8)\}$, maintain $L_1$, $L_2$ and $P$

$L_1’ | s_3(80) | s_4(90) | s_5(95) | s_1(100) | s_2(140) |

$L_2’ | s_4(5) | s_1(6) | s_5(8) | s_2(10) | s_2(11) |

$P’ | 0 | 3 | 5 | 4 | 0 | 1 | 0 | 7$ (permutation array)

$B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0$ (bit-array) new_pos = 4

(3) Visit tuples w.r.t. $L_2’$

(a) Set bits as visited before new_pos

$B | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0$ (for $s_4$ and $s_1$)

(b) visit the new insertion $s_5$

• $\rightarrow$

$B | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0$ Output: $(s_5, s_1)$

$\Rightarrow | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0$

(c) visit $s_3$ (only compare with new insertion $s_5$)

• $\rightarrow$

$B | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0$ Output: $(s_3, s_5)$

$\Rightarrow | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0$

(d) visit $s_2$ (only compare with new insertion $s_5$)

• $\rightarrow$

$B | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0$ Output:

$\Rightarrow | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1$

Figure 4.4: Incremental IESelfJoin process for query $Q_p$

a way similar to the sorted merge join by linearly scanning both sorted arrays in $R \oplus \Delta R^+$ and $S \oplus \Delta S^+$ to set the corresponding offsets. Step (3) is to run IEJoin by only outputting results related to $\Delta R^+$ or $\Delta S^+$, i.e., $(\Delta R^+, S)$, $(R, \Delta S^+)$ or $(\Delta R^+, \Delta S^+)$. If insertions are only $\Delta R^+$, IEJoin ends by visiting the last element of $\Delta R^+$ in the bit-array. Otherwise, it continues until all tuples in $R \oplus \Delta R^+$ are visited. When
processing both insertions and deletions, results from \((\Delta R^+, \Delta S^-)\) or \((\Delta R^-, \Delta S^+)\) are ignored.

### 4.5 Scalable Inequality Joins

I present a scalable version of IEJoin along the same lines of state-of-the-art general-purpose distributed data processing systems, such as Hadoop’s MapReduce [51] and Spark [52]. My goal is twofold: (i) scale my algorithm to very large input relations that do not fit into the main memory of a single machine and (ii) minimize processing overheads to improve the efficiency even further.

A simple approach for scaling IEJoin is to (i) construct \(k\) data blocks of each input relation, (ii) apply Cartesian product (or self-Cartesian product for a single relation input) on the data blocks, and (iii) run IEJoin (either on a single table or two tables input) on all remaining data block pairs (up to \(k^2\)). This approach suffers from a high processing overhead caused by excessive block replication. In particular, excessive data replication increases the CPU overhead by scheduling a large number of tasks and redundantly processing replicated blocks (i.e., sorting identical blocks in different threads). It also causes high memory overheads where different tasks maintain identical copies of the same data. In distributed settings, this approach additionally causes large network overheads as it transfers identical copies of the data to different workers. A naïve solution would be to reduce the number of blocks by increasing their size. However, very large blocks introduce work imbalance and requires larger memory for each worker.

#### 4.5.1 Scalable IEJoin

I solve the above challenges through efficient pre-processing and post-processing phases that reduce data replication by minimizing the number of required data block pairs. I achieve this by pruning unnecessary data block pairs early before wasting any re-
sources. The pre-processing phase generates space-efficient data blocks for the input relation(s), predicts which pair of data blocks may report query results, and only materializes useful pairs of data blocks. IEJoin, in its scalable version, returns the join results as a pair of tupleIDs instead of returning the actual tuples. It is the responsibility of the post-processing phase to materialize the final results by resolving the tupleIDs into actual tuples. I use the internal tupleIDs of existing systems to uniquely identify different tuples. I summarize in Algorithm 7 the implementation of the scalable algorithm when processing two input tables.

**Pre-processing.** After assigning unique tupleIDs to each input tuple (lines 2-3), the pre-processing step globally sorts each relation on a common attribute of one of the IEJoin predicates (i.e., *salary* in $Q_1$). Then, it partitions each sorted relation to $k = \lceil \frac{M}{b} \rceil$ equally-sized partitions, where $M$ is the relation input size and $b$ is the default block size (lines 4-5). Note that global sorting before partitioning maximizes data locality within partitions, which in turn decreases the overall runtime. This is because global sorting partially answers one of the inequality join conditions, where it physically moves tuples closer to their candidate pairs. In other words, global sorting increases the efficiency of block pairs that generate results, while block pairs that do not produce results can be filtered out before actually processing them. After that, for each sorted partition, I generate a single data block that stores only the attribute values referenced in the join conditions in a list. Following the semi-join principle, these data blocks do not store the entire tuples, thus, allowing reduction of the memory, disk I/O, and network overheads. I also extract metadata that contain the block ID and the min/max values of each referenced attribute value from each data block (lines 6-11). Then, I create $n_{MT1} \times m_{MT2}$ virtual block combinations and filter out block combinations with non-intersecting min-max values since they do not produce results (lines 12-17). Figure 4.5 illustrates the pre-processing of two relations, $R$ and $S$. It starts by sorting and partitioning $R$ into three blocks and $S$ into two
**Algorithm 7: Scalable IEJoin**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>//Pre-processing</td>
</tr>
<tr>
<td>2</td>
<td>foreach tuple ( r \in t_1 ) and ( t_2 ) do</td>
</tr>
<tr>
<td>3</td>
<td>( r \leftarrow ) global unique ID</td>
</tr>
<tr>
<td>4</td>
<td>DistT1 ( \leftarrow ) sort ( t_1 ).X and partition to ( n ) blocks</td>
</tr>
<tr>
<td>5</td>
<td>DistT2 ( \leftarrow ) sort ( t_2 ).X' and partition to ( m ) blocks</td>
</tr>
<tr>
<td>6</td>
<td>for ( (i \leftarrow 1 ) to ( n) ) do</td>
</tr>
<tr>
<td>7</td>
<td>( D_{1_i} \leftarrow ) all ( X ) and ( Y ) values in ( DistT_{1_i} )</td>
</tr>
<tr>
<td>8</td>
<td>( MT_{1_i} \leftarrow ) min and max of ( X ) and ( Y ) in ( D_{1_i} )</td>
</tr>
<tr>
<td>9</td>
<td>for ( (j \leftarrow 1 ) to ( m) ) do</td>
</tr>
<tr>
<td>10</td>
<td>( D_{2_j} \leftarrow ) all ( X' ) and ( Y' ) values in ( DistT_{2_j} )</td>
</tr>
<tr>
<td>11</td>
<td>( MT_{2_j} \leftarrow ) min and max of ( X' ) and ( Y' ) in ( D_{2_j} )</td>
</tr>
<tr>
<td>12</td>
<td>Virt ( \leftarrow MT_{1} \times MT_{2} )</td>
</tr>
<tr>
<td>13</td>
<td>forall ( (MT_{1_i},MT_{2_j}) ) pairs ( \in Virt ) do</td>
</tr>
<tr>
<td>14</td>
<td>if ( MT_{1_i} \cap MT_{2_j} ) then</td>
</tr>
<tr>
<td>15</td>
<td>( (MT_{1_i},MT_{2_j}) \leftarrow (D_{1_i},D_{2_j}) )</td>
</tr>
<tr>
<td>16</td>
<td>else</td>
</tr>
<tr>
<td>17</td>
<td>Remove ( (MT_{1_i},MT_{2_j}) ) from Virt</td>
</tr>
<tr>
<td>18</td>
<td>//IEJoin function</td>
</tr>
<tr>
<td>19</td>
<td>forall block pairs ( (D_{1_i},D_{2_j}) ) ( \in Virt ) do</td>
</tr>
<tr>
<td>20</td>
<td>( TupleIDResult \leftarrow ) IEJoin( (Q,D_{1_i},D_{2_j}) )</td>
</tr>
<tr>
<td>21</td>
<td>//Post-processing</td>
</tr>
<tr>
<td>22</td>
<td>forall tupleID pairs ( (v,w) ) ( \in TupleIDResult ) do</td>
</tr>
<tr>
<td>23</td>
<td>( tuple_v \leftarrow ) tuple id ( v ) in DistT1</td>
</tr>
<tr>
<td>24</td>
<td>( tuple_w \leftarrow ) tuple id ( w ) in DistT2</td>
</tr>
<tr>
<td>25</td>
<td>( t_{out} \leftarrow (tuple_v,tuple_w) )</td>
</tr>
</tbody>
</table>

blocks. It then generates the metadata blocks and executes Cartesian product on all metadata blocks. Next, it removes all non-overlapping blocks. Finally, it recovers the original content for all overlapping metadata blocks, i.e., \( \mu R_1 \) recovers its data from block \( R_1 \).

**IEJoin.** I now have a list of overlapping block pairs. I simply run an independent IEJoin for each of these pair blocks in parallel. Specifically, I merge the attribute values in \( D_1 \) and \( D_2 \) and run IEJoin over the merged block. The permutation and
bit arrays generation are similar to the centralized version. However, the scalable IEJoin does not have access to the actual relation tuples. Therefore, each parallel IEJoin instance outputs a pair of tupleIDs that represents the joined tuples (lines 19-20).

Post-processing. In the final step, I materialize the result pairs by matching each tupleID-pair from the output of the distributed IEJoin with the tupleIDs of DistT1 and DistT2 (lines 22-25). I run this post-processing phase in parallel, as a scalable hash join based on the tupleIDs, to speed up the materialization of the final join results.

4.5.2 Multithreaded and Distributed IEJoin

The scalable solution I just described can be adapted to a multithreaded setting as follows. For the pre-processing phase, I sort and partition inputs in parallel. If the
input does not fit into memory, I apply external sorting. Next, block metadata are extracted by all threads, while each thread examines an independent set of block metadata pairs to eliminate non-overlapping blocks in parallel. The remaining block metadata pairs are then materialized, from either a cached copy in memory or from disk, into block pairs. Each thread then applies the IEJoin algorithm on a different block pair to generate partial join results. Once the complete result is computed, I materialize the output using a multi-threaded hash-based join. In a distributed setting, I follow the same process, but I use compute nodes as the main processing units instead of threads. I discuss in Chapter 4.6.2 my distributed version on top of Spark SQL.

4.6 Integration into Existing Systems

I describe the integration of my algorithms into four existing systems: PostgreSQL, a popular open-source DBMS; Spark SQL, a popular SQL-like engine on top of Spark; BigDANSING, my big data cleansing system; and Rheem [109, 110, 130], a cross-platform system for big data analytics.

4.6.1 PostgreSQL

PostgreSQL processes queries in three stages: parsing, planning, and execution. Parsing extracts relations and predicates and creates query parse trees.

Planning creates query plans and invokes the query optimizer to select a plan with the smallest estimated cost. Execution runs the selected plan and emits the output.

**Parsing and Planning.** PostgreSQL uses merge and hash join operators for equijoins and a naïve nested loop for inequality joins. PostgreSQL looks for the most suitable join operator for each join predicate. I extend this check to verify that a join is IEJoin-able by checking if a predicate contains a scalar inequality operator. If it is the case, I save the operator’s oid in the data structure associated with the
predicate. For each operator and ordered pair of relations, I create a list of predicates that the operator can handle. For example, two equality predicates over the same pair of relations are associated to one hash join.

In the presence of inequality joins, I make sure that the PostgreSQL optimizer chooses my IEJoin algorithm, while optimizations for other operators are done by a PostgreSQL optimizer on top of the optimized IEJoin.

Next, the Planner estimates the execution cost for possible join plans. In the presence of both equality and inequality joins, the optimizer will delay all inequality joins as they are usually less selective than equality joins. More specifically, every node in the plan has a base cost, which is the cost of executing the previous nodes, plus the cost for the actual node. I added a cost function for my operator; it is evaluated as the sum of the cost for sorting inner and outer relations, CPU cost for evaluating all output tuples (approximated based on PostgreSQL’s default inequality joins estimation), and the cost of evaluating additional predicates for each tuple (i.e., the ones that are not involved in the actual join). Next, PostgreSQL selects the plan with the lowest cost.

**Execution.** The executor stores incoming tuples from outer and inner relations into arrays of type `TupleTableSlot`, which is PostgreSQL’s default data structure that stores the relation tuples. These copies of the tuples are required as PostgreSQL may not have the content of the tuple at the same pointer location when the tuple is sent for the final projection. This step is a platform-specific overhead that is required to produce an output. The outer relation (of size $N$) is parsed first, followed by the inner relation (of size $M$). If the inner join data is identical to the corresponding outer join data (self-join), I drop the inner join data and the data structure has size $N$ instead of $2N$. If there are more than two IEJoin predicates, then I follow the procedure explained in Chapter 4.3.2, i.e., I pass to the algorithm the pair of predicates with the highest selectivity.
I illustrate in Table 4.2 the data structure and the permutation array computation with an example for self-join $Q_p$. I initialize the data structure with an index ($idx$) and a copy of the attributes of interest ($time$ and $cost$ for $Q_p$). Next, I sort the data on the first predicate ($time$) using system function $qsort$ with special comparators (defined in Algorithm 3) to handle cases where two values for a predicate are equal. The result of the first sort is reported at the left-hand side of Table 4.2. The last column ($pos$) is now filled with the ordering of the tuples according to this sorting. As a result, I create a new array to store the index values for the first predicate. I use this array to select tuple IDs at the time of projecting tuples. The tuples are then ordered again according to the second predicate ($cost$), as reported in the right-hand side of Table 4.2. After the second sorting, the new values in $pos$ are the values for the permutation array.

Finally, I create and traverse a bit-array $B$ of size $(N + M)$ ($N$ in case of self-join) along with a bitmap, as discussed in Chapter 4.2.3. If the traversal finds a set bit, the corresponding tuples are sent for projection.

Predicates, not selected by the optimizer in the case of multi-predicate IEJoin, are evaluated at this stage and, if the conditions are satisfied, tuples are projected.

### 4.6.2 Spark SQL

Spark SQL [131] allows users to query structured data on top of Spark [52]. It stores the input data as a set of in-memory Resilient Distributed Datasets (RDD). Each RDD is partitioned into smaller cacheable blocks, where each block fits in the memory.
of a single machine. Spark SQL takes as input the datasets location(s) in HDFS and an SQL query, and outputs an RDD that contains the query result. The default join operation in Spark SQL is inner join. When passing a join query to Spark SQL, the optimizer searches for equality join predicates that can be used to evaluate the inner join operator as a hash-based physical join operator. If there are no equality join predicates, the optimizer translates the inner join physically to a Cartesian product followed by a selection predicate.

I implemented the distributed version of IEJoin as a new Spark SQL physical join operator. To make the optimizer aware of the new operator, I added a new rule to recognize inequality conditions. The rule passes all inequality conditions to the IEJoin operator. If the operator receives more than two inequality join conditions, it deploys the IEJoin optimizer to find the two highest selective inequality conditions. It executes the join using such conditions and evaluates the rest of the join conditions as a post selection operation on the output. Similar to the PostgreSQL case, in the presence of both equality and inequality joins, it orders inequality joins after all equality joins. The distributed operator utilizes Spark RDD operators to run both the IEJoin and its optimizer. As a result, distributed IEJoin depends on Spark’s default memory management to partition and store the user’s input relation. If the result does not fit in the memory of a single machine, I temporarily store the result into HDFS. After all IEJoin instances finish writing into HDFS, the distributed operator passes the HDFS file to Spark, which constructs a new RDD of the result and passes it to Spark SQL.

Figures 4.6 and 4.7 show how the distributed IEJoin is processed in Spark. First, I globally sort the two relations using Spark RDD sort (Figure 4.6(a)). Next, I generate a set of distributed data blocks for each relation through the RDD mapPartitionsWithIndex() function (Figure 4.6(b)). As described in Chapter 4.5, the block transformation does not store the actual tuples; it only stores the attributes
in the join predicates. I then transform the data blocks into metadata blocks using their statistics (Figure 4.6(c)). Afterwards, I apply a Cartesian product on the block metadata of \( R \) and \( S \) and remove non-overlapping block metadata through the RDD \texttt{filter()} operator (Figure 4.6(d)). Next, I join the remaining blocks’ metadata with the original data blocks to recover their content (Figure 4.6(e)) through two RDD \texttt{join()} operators; one for each input relation. I then apply an independent instance
of IEJoin on every block pair in parallel by using RDD `flatMapToPair()` operator (Figure 4.7(a)). Finally, I join the IEJoin result with the original relations $R$ and $S$, using two RDD `join()` operators, to recover the full attribute information of the result (Figure 4.7(b)).

### 4.6.3 BigDansing

In BigDansing, OCJoin (Chapter 3.3.3) is used to detect violations when dealing with rules that require inequality joins. For example, when the rule states that given a dataset for employees in the same State, for every two distinct individuals, the one earning a lower *salary* should have a lower tax *rate*. OCJoin is a variation of distributed sort-merge join which performs better than the distributed cross product in BigDansing; however, it is only efficient for very small datasets (in the order of 100K tuples only).
I integrate IEJoin within BigDansing to further improve the detection process for such rules and scale to much larger datasets. To achieve this, I implement the distributed IEJoin on Spark, as a set of RDD transformations, and add it to BigDansing as a new enhancer and execution operator for Spark. After that, I modify BigDansing’s logical plan translator to consider the new enhancer whenever the input rule requires inequality joins. Notice that the IEJoin execution operator is similar to its implementation in Spark SQL discussed in Chapter 4.6.2. I show in Figure 4.8 a summary of BigDansing’s modified operators for Spark with IEJoin.

### 4.6.4 Rheem

Rheem is an open-source big data processing framework that provides independence from and interoperability among existing data processing platforms. I implement IEJoin in Rheem to provide IEJoin as a service for applications that require IEJoin without the overhead of porting their application to either PostgreSQL or Spark SQL. More specifically, (i) I extend the Rheem framework in order to expose the new join as a physical operator for users, (ii) I add the centralized IEJoin as an execution operator for Java, and (iii) I implement the distributed version of IEJoin as an execution operator.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of tuples</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employees</td>
<td>10K – 500M</td>
<td>300KB – 17GB</td>
</tr>
<tr>
<td>Employees2</td>
<td>1B – 8B</td>
<td>34GB – 287GB</td>
</tr>
<tr>
<td>Events</td>
<td>10K – 500M</td>
<td>322KB – 14GB</td>
</tr>
<tr>
<td>Events2</td>
<td>1B – 6B</td>
<td>32GB – 202GB</td>
</tr>
<tr>
<td>MDC</td>
<td>24M</td>
<td>2.4GB</td>
</tr>
<tr>
<td>Cloud</td>
<td>470M</td>
<td>28.8GB</td>
</tr>
<tr>
<td>Grades</td>
<td>100K</td>
<td>1.7MB</td>
</tr>
</tbody>
</table>

Table 4.3: Size of the datasets

operator for Spark. As a result, Rheem users can switch between the centralized IEJoin and distributed IEJoin in their application transparently without changing their code.

4.7 Experimental Study

I evaluate IEJoin along several dimensions: (i) Effect of sorting and caching; (ii) IEJoin in a centralized environment; (iii) Query optimization techniques; (iv) Incremental algorithms; and (v) IEJoin in a distributed environment.

4.7.1 Datasets

I used both synthetic and real-world data (summarized in Table 4.3) to evaluate my algorithms.

(1) **E</div>
large output files.

(2) **Events** is a synthetic dataset that contains start and end time information for a set of independent events. Each event contains the name of the event, event ID, the number of attending people, and the sponsor ID. To make sure I generate output for a given query, I extended end values for 10% random events. **Events2** contains larger datasets with up to 6 Billion records and 0.001% extended random events.

(3) **Mobile Data Challenge (MDC)** is a 50GB real dataset [132, 133] that contains behavioral data of nearly 200 individuals collected by Nokia Research (https://www.idiap.ch/dataset/mdc). The dataset contains physical locations, social interactions, and phone logs of the participating individuals.

(4) **Cloud** [57] is a real dataset that contains cloud reports from 1951 to 2009, through land and ship stations (ftp://cdiac.ornl.gov/pub3/ndp026c/).

(5) **Grades** is a synthetic dataset for testing the not equal (≠) join predicate. It is composed of a list of students with attributes id, name, gender, grade, and age. I encode gender values by using numeric values to be able to compare them using conditions less than (<) and greater than (>). The dataset size is fixed (100K tuples), but I vary the ratio of female students between 0.01% and 50% to diversify the selectivity of the query.

### 4.7.2 Algorithms

I compare my algorithms with several centralized as well as distributed algorithms

**Centralized Systems.** For my centralized setting, I use the following systems:

(1) **C++ IEJOIN.** This is a standalone C++ implementation of IEJoin to run experiments that cannot be executed within a DBMS. For example, I use this implementation to evaluate the incremental experiments of IEJoin (Chapter 4.7.8). The
implementation uses optimized data structures from the Boost library\(^2\) to maximize the performance gain of IEJoin through a faster array scanning.

(2) PG-IEJOIN. I implemented IEJoin inside PostgreSQL v9.4, as discussed in Chapter 4.6.1. I compare it against the baseline systems below.

(3) PG-ORIGINAL. I used PostgreSQL v9.4 as a baseline. I tuned it with pgtune to maximize the benefit from the large main memory.

(4) PG-BTree & PG-GiST. I also used two variants of PostgreSQL using indices. PG-BTree uses a B-tree index for each attribute in a query. PG-GiST uses the GiST access method in PostgreSQL, which considers arbitrary indexing schemes and automatically selects the best technique for the input relation.

(5) MONETDB. I used MonetDB Database Server Toolkit v1.1 (Oct2014-SP2), an open-source column-oriented database, in a disk partition of size 669GB.

(6) DBMS-X. I used a leading commercial centralized relational database.

**Distributed Systems.** I used the following systems:

(1) SPARK SQL-IEJOIN. I implemented IEJoin inside Spark SQL v1.0.2 (https://spark.apache.org/sql/) as detailed in Chapter 4.6.2. I evaluated the performance of IEJoin against the baseline systems below.

(2) SPARK SQL & SPARK SQL-SM. Spark SQL is the default implementation in Spark SQL. Spark SQL-SM is an optimized version based on distributed sort-merge join (Chapter 4.6.3) which is identical to BigDancing’s OCJoin in Chapter 3.3.3. I also improve the above method by pruning the non-overlapping partitions to be joined.

\(^2\)http://www.boost.org/
DPG-BTree & DPG-GiST. I used a commercial version of PostgreSQL with distributed query processing. This allows me to compare Spark SQL-IEJOIN to a distributed version of PG-BTree and PG-GiST.

4.7.3 Queries

I evaluate my algorithms from different perspectives and using several queries with inequality join conditions. It is worth noting that my main goal in the experiments is to show the value of optimizing inequality queries using my approach irrespective of the presence of other operators.

For my first experiment, I used the following self-join query over Employees to find violations of a data quality rule [6]:

\[ Q_1 : \text{SELECT r.id, s.id FROM Employees r, Employees s} \]
\[ \text{WHERE r.salary < s.salary AND r.tax > s.tax;} \]

The query returns a set of employee pairs, where one employee earns a higher salary than the other, but pays less tax. To make sure that I generate output for \( Q_1 \), I selected 10% random tuples and increased their tax values. I also used a self-join query that collects pairs of overlapping events (in the Events dataset):

\[ Q_2 : \text{SELECT r.id, s.id FROM Events r, Events s} \]
\[ \text{WHERE r.start \leq s.end AND r.end \geq s.start AND r.id \neq s.id;} \]

I extended end values for 10% random events to make sure I generate output for \( Q_2 \). Throughout all of my experiments, I either use these two queries or slightly modified versions thereof. For example, I added a third join predicate to \( Q_1 \) over age to get \( Q'_1 \) and extended \( Q_1 \) and \( Q_2 \) to get \( Q_{mw} \) to study how my algorithms deal with multi-predicate conditions and with multi-way joins, respectively.
$Q_1$ : SELECT r.id, s.id FROM Employees r, Employees s
WHERE r.salary < s.salary AND r.tax > s.tax AND r.age > s.age;

$Q_{mw}$: SELECT count(*) FROM R r, S s, T t, V v, W w
WHERE r.salary > s.salary AND r.tax < s.tax
AND s.start < t.end AND s.end > t.start
AND t.salary > v.salary AND t.tax < v.tax
AND v.start < w.end AND v.end > w.start;

I also used slightly modified versions of $Q_1$ and $Q_2$ for my comparison with baselines using indexes. This is because although $Q_1$ and $Q_2$ appear to be similar, they require different data representation to be indexed with GiST. The inequality attributes in $Q_1$ are independent, each condition forms a single open interval, while in $Q_2$ they are dependent, together they form a single closed interval. Thus, I convert salary and tax attributes into a single geometric point data type SalTax to get $Q_{1i}$. Similarly for $Q_2$, I convert start and end attributes into a single range data type StartEnd to get $Q_{2i}$.

$Q_{1i}$: SELECT r.id, s.id FROM Employees r, Employees s
WHERE r.SalTax >\^ s.SalTax AND r.SalTax \gg s.SalTax;

$Q_{2i}$: SELECT r.id, s.id FROM Events r, Events s
WHERE r.StartEnd && s.StartEnd AND r.id \neq s.id;

In the rewriting of these queries for PG-GiST, operator “\^” corresponds to “is above?”, operator “\gg” means “is strictly right of?”, and operator “&&” indicates “overlap?”. For geometric and range types, GiST uses a Bitmap index to optimize its data access with large datasets.

In addition to the above two main queries, I used $Q_3$ to evaluate my algorithms when producing different output sizes. This query looks for all persons that are close to a shop up to a distance $c$ along the x-axis ($xloc$) and the y-axis ($yloc$):
\(Q_3:\) SELECT s.name, p.name FROM Shops s, Persons p
WHERE s.xloc - c < p.xloc AND s.xloc + c > p.xloc
AND s.yloc - c < p.yloc AND s.yloc + c > p.yloc;

I also used a self-join query \(Q_4\), similar to \(Q_3\), to compute all stations within distance \(c = 10\) for every station. Since the runtime in \(Q_3\) and \(Q_4\) is dominated by the output size, I mostly used them for scalability analysis in the distributed case.

Furthermore, I used query \(Q_5\) for my not equal join predicates experiment.

\(Q_5:\) SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender \neq s.gender AND r.grade > s.grade;

Notice that PostgreSQL executes \(Q_5\) with nested loops. In my solution, the query is rewritten to \(Q'_5\).

\(Q'_5:\) SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender < s.gender AND r.grade > s.grade
UNION ALL
SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender > s.gender AND r.grade > s.grade;

As attribute gender in \(Q_5\) has only two distinct values, it might not provide the complete picture of the performance of \((\neq)\) IEJoins. Thus, I additionally used \(Q_6\) and \(Q_7\) to analyze the effect of non-binary attributes on IEJoin with \((\neq)\) as the join predicates.

\(Q_6:\) SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.age \neq s.age AND r.grade > s.grade;

\(Q_7:\) SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender \neq s.gender AND r.grade \neq s.grade;
### Table 4.4: Bitmaps on 10M tuples (Events data)

<table>
<thead>
<tr>
<th>Chunk size (bits)</th>
<th>C++ (Sec)</th>
<th>PostgreSQL (Sec)</th>
<th>Spark SQL (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt; 1 day</td>
<td>&gt; 1 day</td>
<td>&gt; 1 day</td>
</tr>
<tr>
<td>64</td>
<td>2333</td>
<td>3139</td>
<td>1623</td>
</tr>
<tr>
<td>256</td>
<td>558</td>
<td>817</td>
<td>896</td>
</tr>
<tr>
<td>1024</td>
<td>158</td>
<td>242</td>
<td>296</td>
</tr>
<tr>
<td>4096</td>
<td>81</td>
<td>117</td>
<td>158</td>
</tr>
<tr>
<td>16384</td>
<td>139</td>
<td>142</td>
<td>232</td>
</tr>
</tbody>
</table>

### 4.7.4 Setup

For the centralized evaluation, I used a Dell Precision T7500 with two 64-bit quad-core Intel Xeon X5550 and 58GB RAM. For my distributed experiments, I used a cluster of 17 Shuttle SH55J2 machines (1 master with 16 workers) with Intel i5 processors with 16GB RAM, and connected to a high-end 1 Gigabit switch. For both settings, all arrays are stored in memory.

### 4.7.5 Parameter Setting

I start my experimental evaluation by showing the effect of the two optimizations (Chapter 4.2.3), as well as the effect of global sorting (Chapter 4.5).

**Bitmap.** Bitmaps are used when big array scanning is expensive. The chunk size is an optimization parameter that is machine-dependent. I run query $Q_2$ on 10M tuples with size 322MB to show the performance gain of using a bitmap based on three different implementations of IEJoin: the centralized C++, the C implementation of PostgreSQL, and the Java implementation of Spark SQL. For Spark SQL, I only ran a single instance of IEJoin in a centralized setting.

Results are shown in Table 4.4. Intuitively, the larger the chunk size, the better. However, a very large chunk size defeats the purpose of using bitmaps to reduce the bit-array scanning overhead. The experiment shows that the performance gain is 3X
Table 4.5: Bitmaps on 10M tuples with max scan index optimization (Events data)

<table>
<thead>
<tr>
<th>Max Index?</th>
<th>C++ (Sec)</th>
<th>PostgreSQL (Sec)</th>
<th>Spark SQL (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>81</td>
<td>117</td>
<td>158</td>
</tr>
<tr>
<td>Yes</td>
<td>23</td>
<td>47</td>
<td>46</td>
</tr>
</tbody>
</table>

between 256 bits and 1,024 bits and around 1.8X between 1,024 bits and 4,096 bits. Larger chunk sizes show worse performance, as shown with chunk size of 16,384 bits. The experiment in Table 4.5 shows the performance of the three implementations of IEJoin when adding the **max scan index** optimization to the bitmap. This index optimization improves the performance of IEJoin around 3.5 times compared to the best results achieved by the bitmap in Table 4.4.

**Union arrays.** To study the impact of the union optimization, I run IEJoin with and without the union array using 10M tuples in the **Events** dataset. I collect the following statistics with SPARK SQL-IEJOIN, as shown in Table 4.6: (i) L1 data caches (dcache), (ii) last level cache (LLC), and (iii) data translation lookaside buffer (dTLB). The optimized algorithm with union arrays is 2.6 times faster than the original one. The performance gain in the optimized version is due to the lower number of cache loads and stores (L1-dcache-loads, L1-dcache-stores, dTLB-loads and TLB-stores), which is 2.7 to 3 times smaller than the original algorithm. This behavior is expected since the optimized IEJoin has fewer arrays w.r.t. the original version.

**Global sorting on distributed IEJoin.** As presented in Algorithm 7, the distributed version of my algorithm applies global sorting in the pre-processing phase (lines 6-7). I report in Table 4.7 a detailed performance comparison for $Q_1$ and $Q_2$ with and without global sorting on 100M tuples from the Employees and Events datasets. The pre-processing time includes data loading from HDFS, global sorting, partitioning, and block-pairs materialization. One may think that the global sorting
<table>
<thead>
<tr>
<th>Parameter (M/sec)</th>
<th>IEJoin (union)</th>
<th>IEJoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache-references</td>
<td>6.5</td>
<td>8.4</td>
</tr>
<tr>
<td>cache-references-misses</td>
<td>3.9</td>
<td>4.8</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>459.9</td>
<td>1,240.6</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>8.7</td>
<td>10.9</td>
</tr>
<tr>
<td>L1-dcache-stores</td>
<td>186.8</td>
<td>567.5</td>
</tr>
<tr>
<td>L1-dcache-store-misses</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>L1-dcache-prefetches</td>
<td>4.9</td>
<td>7.0</td>
</tr>
<tr>
<td>L1-dcache-prefetches-misses</td>
<td>2.2</td>
<td>2.7</td>
</tr>
<tr>
<td>LLC-loads</td>
<td>5.1</td>
<td>6.6</td>
</tr>
<tr>
<td>LLC-load-misses</td>
<td>2.9</td>
<td>3.7</td>
</tr>
<tr>
<td>LLC-stores</td>
<td>3.8</td>
<td>3.7</td>
</tr>
<tr>
<td>LLC-store-misses</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>LLC-prefetches</td>
<td>3.1</td>
<td>4.1</td>
</tr>
<tr>
<td>LLC-prefetch-misses</td>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>dTLB-loads</td>
<td>544.4</td>
<td>1,527.2</td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>0.9</td>
<td>1.6</td>
</tr>
<tr>
<td>dTLB-stores</td>
<td>212.7</td>
<td>592.6</td>
</tr>
<tr>
<td>dTLB-store-misses</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>Total time (sec)</strong></td>
<td><strong>125</strong></td>
<td><strong>325</strong></td>
</tr>
</tbody>
</table>

Table 4.6: Cache statistics on 10M tuples (Events data)

impairs the performance of distributed IEJoin as it shuffles data through the network. However, global sorting improves the performance of the distributed algorithm by 2.4 to 2.9 times. More specifically, the runtime for the pre-processing phase with global sorting is at least 30% faster compared to the case without global sorting. Moreover, I also note that the time required by IEJoin is one order of magnitude faster when using global sorting. This is because global sorting enables filtering out block-pair combinations that do not generate results. This greatly reduces the network overhead and increases memory locality in the block combinations that are passed to my algorithm.

Based on the above experiments, in the following tests I used 1,024 bits as the default chunk size, the max scan index optimization, union arrays, and global sorting for distributed IEJoin.
<table>
<thead>
<tr>
<th>Query</th>
<th>Pre-process</th>
<th>IEJoin</th>
<th>Post-process</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With global sorting (Seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>632</td>
<td>162</td>
<td>519</td>
<td>1,313</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>901</td>
<td>84</td>
<td>391</td>
<td>1,376</td>
</tr>
<tr>
<td></td>
<td>Without global sorting (Seconds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>1,025</td>
<td>1,714</td>
<td>426</td>
<td>3,165</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>1,182</td>
<td>1,864</td>
<td>349</td>
<td>3,395</td>
</tr>
</tbody>
</table>

Table 4.7: IEJoin time analysis on 100M tuples and 6 workers

### 4.7.6 Single-node Experiments

In this set of experiments, I study the efficiency of IEJoin on datasets that fit the main memory of a single compute node and compare its performance with alternative centralized systems.

**IEJoin vs. baseline systems.** Figure 4.9 shows the results for queries $Q_1$ on Employees dataset and $Q_2$ on Events dataset in a centralized environment using 10K, 50K and 100K tuples. The $x$-axis represents the input size in terms of the number of tuples and the $y$-axis represents the corresponding running time in seconds. The figure reports that PG-IEJoin outperforms all baseline systems by more than one order of magnitude for both queries, and for every reported dataset input size. In particular, PG-IEJoin is up to more than three (resp., two) orders of magnitude faster than PG-ORIGINAL and MONETDB (resp., DBMS-X). Clearly, the baseline systems cannot compete with PG-IEJoin since they all use the classic Cartesian product followed by a selection predicate. In fact, this is the main reason why they cannot run for bigger datasets.

**IEJoin vs. indexing.** I now consider the two variants of PostgreSQL, PG-BTREE & PG-GiST, to evaluate the efficiency of my algorithm on bigger datasets. I again run $Q_1$ and $Q_2$ on 10M and 50M records using both Employees and Events datasets. Figure 4.10 shows the results. In both experiments, IEJoin is more than one order of magnitude faster than PG-GiST. In fact, IEJoin is more than three times faster than
the GiST indexing time alone. I stopped PG-BTree after 24 hours of runtime. My algorithm performs better than these two baseline indices because it better utilizes the memory locality.

**Memory consumption.** The memory consumption for MonetDB increases exponentially with the input size. For example, MonetDB uses 419GB for an input dataset with only 200K records. In contrast to MonetDB, IEJoin makes better use of the memory. Table 4.8 shows that IEJoin uses around 200MB for $Q_1$ and $Q_2$ for an input dataset of 200K records, while MonetDB requires two orders of magnitude more memory. In Table 4.8 I also report the overall memory used by sorted attribute arrays, permutation arrays, and the bit-array. Moreover, although IEJoin requires 8.2GB of memory for an input dataset of 10M records, it runs to completion in about 8 hours (28,928 seconds) for a dataset, producing more than 50 billion output records.

**Time breakdown.** I further analyze the breakdown time of IEJoin on Employees and Events datasets with 50M tuples. Table 4.9 shows that, by excluding the time required to load the dataset into memory, scanning the bit-array takes only 10% of the overall execution time in C++ IEJOIN, 5% in PG-IEJOIN, and 40% in Spark SQL-IEJOIN, while the rest is mainly for sorting. This shows the high efficiency of my
IEJoin with the not equal (≠) join predicate. I tested the performance of IEJoin with the not equal join predicate with $Q_5$ on the Grades dataset. PG-ORIGINAL runs the query by using nested loops, while PG-IEJOIN uses two instances of IEJoin to process $Q'_5$. As shown in Figure 4.11(a), PG-IEJOIN is from four times to two orders of magnitude faster than PG-ORIGINAL. Note that the query output becomes larger while increasing the Female frequency in the dataset.

I also tested the effect of a non-binary attribute by using queries $Q_5$, $Q_6$, and $Q_7$. For PG-IEJOIN, the queries were transformed into a union of inequality joins as discussed before. Figure 4.11(b) shows that running PG-IEJOIN on $Q_5$ and $Q_7$ is an order of magnitude faster than PG-ORIGINAL, while on $Q_6$ it is four times faster. This is because the join attribute in $Q_5$ and one of the join attributes in $Q_7$ generate far less results than $Q_6$. In fact, the output of $Q_6$ is five times higher than the one of $Q_5$ and three times higher than the one of $Q_7$. This difference on $Q_6$ is expected since both attributes of the not equal join predicates are not binary.

IEJoin vs. cache-efficient Cartesian product. I further push my evaluation to better highlight the memory locality efficiency on $Q_1$ using Employees dataset.
<table>
<thead>
<tr>
<th>Query</th>
<th>Input</th>
<th>Output</th>
<th>Time(secs)</th>
<th>Mem(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>100K</td>
<td>9K</td>
<td>0.30</td>
<td>0.1</td>
</tr>
<tr>
<td>$Q_1$</td>
<td>200K</td>
<td>1.1K</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>$Q_1$</td>
<td>1M</td>
<td>29K</td>
<td>2.79</td>
<td>0.9</td>
</tr>
<tr>
<td>$Q_1$</td>
<td>10M</td>
<td>3M</td>
<td>27.64</td>
<td>8.8</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>100K</td>
<td>0.2K</td>
<td>0.34</td>
<td>0.1</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>200K</td>
<td>0.8K</td>
<td>0.65</td>
<td>0.2</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>1M</td>
<td>20K</td>
<td>3.38</td>
<td>0.9</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>10M</td>
<td>2M</td>
<td>59.6</td>
<td>9.7</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>100K</td>
<td>6M</td>
<td>2.8</td>
<td>0.1</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>200K</td>
<td>25M</td>
<td>10.6</td>
<td>0.2</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>1M</td>
<td>0.4B</td>
<td>186</td>
<td>0.9</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>10M</td>
<td>50.5B</td>
<td>28,928</td>
<td>8.2</td>
</tr>
</tbody>
</table>

Table 4.8: Runtime and memory usage (PG-IEJOIN)

<table>
<thead>
<tr>
<th>Query</th>
<th>Data reading</th>
<th>Data sorting</th>
<th>Bit-array scanning</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C++ IEJOIN.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>82</td>
<td>31</td>
<td>4</td>
<td>117</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>85</td>
<td>31</td>
<td>3</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>PG-IEJOIN.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>46</td>
<td>94</td>
<td>4</td>
<td>146</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>48</td>
<td>238</td>
<td>24</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>SPARK SQL-IEJOIN (Single Node).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>158</td>
<td>240</td>
<td>165</td>
<td>563</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>319</td>
<td>332</td>
<td>215</td>
<td>866</td>
</tr>
</tbody>
</table>

Table 4.9: Time breakdown on 50M tuples, all times in seconds

I compare the performance of C++ IEJOIN with both naïve and cache-efficient Cartesian product joins for $Q_1$ on datasets that fit the L1 cache (256 KB), L2 cache (1 MB), and L3 cache (8 MB) of the Intel Xeon processor. I used 10K tuples for L1 cache, 40K tuples for L2 cache, and 350K tuples for L3 cache. I do not report results for query $Q_2$ because they are similar to $Q_1$. In Figure 4.12, I see that when the dataset fits in the L1 cache, IEJoin is two orders of magnitude faster than both the cache-efficient Cartesian product and the naïve Cartesian product. Furthermore, as I increase the dataset size for $Q_1$ to be stored at the L2 and L3 caches, I see that IEJoin becomes almost three and four orders of magnitude faster than the Cartesian product.
Figure 4.11: \( Q_5 \) runtime with different Female distributions, and \( Q_5, Q_6 \) and \( Q_7 \) runtimes with 10% Female distribution.

![Graph](image)

Figure 4.12: \( Q_1 \) runtime for data that fits caches.

![Graph](image)

product. This is because of the delays of L2 and L3 caches and the complexity of the Cartesian product.

**Single-node summary.** IEJoin outperforms existing baselines by at least an order of magnitude for two main reasons: it avoids the use of Cartesian product and it exploits memory locality by using memory-contiguous data structures with small footprint. In other words, my algorithm avoids, as much as possible, going to memory to fully exploit the CPU speed.
<table>
<thead>
<tr>
<th>Join Predicates</th>
<th>Estimation on a % sample (number of overlapping blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>salary &amp; tax</td>
<td>501</td>
</tr>
<tr>
<td>salary &amp; age</td>
<td>125250</td>
</tr>
<tr>
<td>tax &amp; age</td>
<td>125250</td>
</tr>
</tbody>
</table>

Table 4.10: $Q'_1$’s selectivity estimation on 50M tuples Employees dataset with low selectivity age attribute

<table>
<thead>
<tr>
<th>Join Predicates</th>
<th>Estimation on a % sample (number of overlapping blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>salary &amp; tax</td>
<td>500</td>
</tr>
<tr>
<td>salary &amp; age</td>
<td>503</td>
</tr>
<tr>
<td>tax &amp; age</td>
<td>124750</td>
</tr>
</tbody>
</table>

Table 4.11: $Q'_1$’s selectivity estimation on 50M tuples Employees dataset with high selectivity age attribute

### 4.7.7 IEJoin Optimizer Experiments

In this section, I evaluate the accuracy of the IEJoin optimizer when dealing with multi-predicate and multi-way queries. I use $Q'_1$ for multi-predicate IEJoin and $Q_{mw}$ for multi-way IEJoin. For the multi-predicate experiment, I evaluate the performance of IEJoin by using the highest selective predicates, found by Algorithm 5, compared to using other predicates. For the multi-way IEJoin, I compare the runtime of the plan generated by Algorithm 6 against other plans.

**Multi-predicate IEJoin.** I evaluate the accuracy of the selectivity estimation algorithm on query $Q'_1$ using C++ IEJOIN. I first use Algorithm 5 to calculate the selectivity estimation for all three predicate pairs using the join attributes salary, tax, and age.

To evaluate the estimation algorithm, I generated two different distributions of attribute age for query $Q'_1$: a low selectivity distribution that generates a large output with salary and tax attributes, and a high selectivity distribution that generates a small output with attribute salary. The low selectivity distribution injects random
noise on the age attribute in each tuple, generating a large output for the IEJoin on \((r\.salary < s\.salary \text{ AND } r\.age > s\.age)\) or on \((r\.tax > s\.tax \text{ AND } r\.age > s\.age)\). For the high selectivity distribution, however, I carefully assign a value proportional to the salary attribute to generate a small output for the IEJoin on \((r\.salary < s\.salary \text{ AND } r\.age > s\.age)\). I show the selectivity estimation for both distributions of the age attribute in Tables 4.10 and 4.11. The tables report selectivity estimations when using input sample of size 1%, 5%, 10%, and 20%. I determine the best join predicate pair for \(Q'_1\) by selecting the predicate pair with the minimum overlapping blocks. According to Tables 4.10 and 4.11, the \((\text{salary}, \text{tax})\) join predicate pair has the highest selectivity in all input samples. Although the difference between predicates pairs \((\text{salary}, \text{tax})\) and \((\text{salary}, \text{age})\) in Table 4.11 is relatively small, I notice that it gets larger as I increase the size of the sample input. To validate this observation, I execute IEJoin on each join predicate pair and compare the output size and runtime values.

I show in Figure 4.13(a) that choosing \((\text{salary}, \text{tax})\) is the right decision. In fact, the output size of \((\text{salary}, \text{tax})\) with the low selectivity age attribute is at least two orders of magnitude lower than the results with other predicates, and it is 50% lower.
than (salary, age) with the high selectivity age attribute. In Figure 4.13(b) I show the runtime difference among predicate pairs with the high selectivity age attribute. Join predicate pair (salary, tax) is orders of magnitude faster than predicate pair (tax, age), but only a couple of seconds faster than predicate pair (salary, age). Although the performance difference between (salary, tax) and (salary, age) pairs are small in the dataset with high selectivity age, the IEJoin query optimizer was able to detect that using (salary, tax) is faster than using (salary, age). I also notice that the runtime for the low selectivity age dataset is orders of magnitude faster with predicate pair (salary, tax) compared with other combinations.

**Multi-way IEJoin.** I test the multi-way IEJoin optimization in Algorithm 6 with the five relations in $Q_{mw}$. Each relation in $Q_{mw}$ contains a random number of tuples from both Employees and Events datasets. I summarize in Table 4.12 the size of each relation, the selectivity estimation (computed with Algorithm 5 on a 1% sample size), and the join result size based on the inequality conditions in $Q_{mw}$. Table 4.12 shows that the selectivity estimation is consistent with the actual join result size; the lower the estimation, the smaller the output size, and the higher the estimation, the larger the output size. Based on Algorithm 6, the optimal plan for $Q_{mw}$ is $(((R \bowtie S) \bowtie T) \bowtie V) \bowtie W$ according to the selectivity estimations in Table 4.12. To evaluate the quality of the optimal plan, I evaluate the performance of all multi-way IEJoin plans for $Q_{mw}$ with PG-IEJoin, and report the results in Table 4.13. Plan $(((R \bowtie S) \bowtie T) \bowtie V) \bowtie W$ that was generated by Algorithm 6 is indeed the fastest one. Although $(R \bowtie S)$ has higher selectivity than $(T \bowtie V)$, evaluating $(T \bowtie V)$ is slightly faster than $(R \bowtie S)$. Indeed $(R \bowtie S)$ produces less results compared to $(T \bowtie V)$ (Table 4.12), but since sorting dominates the performance for IEJoin, as shown in Table 4.9 $(R \bowtie S)$ becomes slightly slower with a larger number of tuples to sort (40M tuples) compared to $(T \bowtie V)$ (38M tuples). Nevertheless, $(((R \bowtie S) \bowtie T) \bowtie V) \bowtie W$ remains the fastest plan because it eliminates more
tuples earlier compared to the rest.

**Optimizer experiments summary.** With only 1% of the data, selectivity estimation in Algorithm 5 is able to accurately distinguish between high and low selective IEJoins. Note that the 1% sample size may not be applicable for other datasets since the minimum sample size depends on the distribution of the join attributes. The performance degradation of not selecting the optimal query execution plan for multi-predicate and multi-way IEJoins varies between 5% and orders of magnitude performance drop. With negligible overhead, my selectivity estimation allows a DBMS to tune its query optimizer to select the optimal plan for multi-predicate and multi-way IEJoins.
4.7.8 Incremental IEJoin Experiments

I developed the incremental algorithm in Chapter 4.4 using C++ IEJOIN and tested it on $Q_1$. Again, I do not report results for $Q_2$ since they were similar to those of $Q_1$. I used five different implementations for the experiments in this section: (1) Non-incremental (my original implementation), (2) incremental non-batched sort-merge ($\Delta Inc$), (3) incremental batched sort-merge ($B-\Delta Inc$), (4) incremental with the packed-memory array (PMA) data structure to dynamically maintain the cached input ($P-\Delta Inc$), and (5) $\sigma Inc$ which is similar to $B-\Delta Inc$ but returns the full output. The three variations of $\Delta Inc$ return results that correspond only to the updates while $\sigma Inc$ generates results from both the updates and the cached input.

**Incremental IEJoin with small updates.** I first study the advantage of my $P-\Delta Inc$ algorithm on small updates. In Figure 4.14, I compare the runtime of $\Delta Inc$, $B-\Delta Inc$, non-incremental, and $P-\Delta Inc$ by using 80M tuples as cached input. I consider $\Delta Inc$ as the baseline for incremental IEJoin. In Figure 4.14, $\Delta Inc$ is twice faster than the non-incremental IEJoin for update of size 1. As the size of the update increases, $\Delta Inc$ slows down because it processes each update individually. To avoid this overhead, I use $P-\Delta Inc$ to increase the efficiency of individual updates and $B-\Delta Inc$ to process updates in a batched fashion. Since the update sizes are relatively small, the runtime of both $B-\Delta Inc$ and the non-incremental algorithm across different update sizes remain constant. Figure 4.14 shows that $B-\Delta Inc$ is always twice faster than the non-incremental algorithm, while $P-\Delta Inc$ is 30\% better than $B-\Delta Inc$ and three times faster than the non-incremental with update of size 1. The processing overhead of $P-\Delta Inc$ is directly proportional to the update size, and its performance declines as the update size increases. Although $P-\Delta Inc$ is up to two orders of magnitude faster than $\Delta Inc$ on higher update sizes, $B-\Delta Inc$ is better than $P-\Delta Inc$ on updates larger than 50. The limitation of $P-\Delta Inc$ on updates larger than 50 is inherited from the design of PMA which works better with individual updates. Note that $P-\Delta Inc$ has
an extra 60% memory overhead, compared to the non-incremental IEJoin, due to the extra empty spaces maintained by the PMA.

**Incremental IEJoin with large updates.** In these experiments, I focus on $B-\Delta Inc$ and $\sigma Inc$ since both $\Delta Inc$ and $P-\Delta Inc$ do not work well with large updates. I show in Figures 4.15(a), 4.15(b), and 4.15(c) a comparison between the runtime (without data loading) of $B-\Delta Inc$ and non-incremental IEJoin with different update and cached input sizes. In Figure 4.15(a), I start with 10M tuples as cached input. $B-\Delta Inc$ is 40% faster on an update of size 1M tuples and has 5 times less output compared to the non-incremental algorithm. For updates larger than 1M, however, I notice that the performance of $B-\Delta Inc$ significantly drops as the difference in the output sizes between the $B-\Delta Inc$ and the non-incremental becomes insignificant. Similar behavior can be observed in Figures 4.15(b) and 4.15(c), with a cached input of 20M and 30M tuples, respectively. $\Delta Inc$ is 50% and 30% faster than the non-incremental algorithm on updates of sizes 1M and 5M tuples, respectively, while its performance drops on updates larger than 5M tuples in both cases. I also notice that $B-\Delta Inc$ on the 1M tuples update has a 10 times smaller output in Figure 4.15(b) and a 15 times smaller output in Figure 4.15(c) compared to the non-incremental algorithm.
From the above three figures, $B$-$\Delta Inc$ clearly shows significant improvement over the non-incremental algorithm (up to 50%) with update sizes that generate significantly smaller output. In this experimental setup, an efficient update size for $B$-$\Delta Inc$ does not exceed 25% of the cached input size.

I further tested the effect of update sizes on disk reading and memory consumption on $B$-$\Delta Inc$ when using 30M tuples as a cached input. Based on Figure 4.16(a), $B$-$\Delta Inc$ gains up to 90% performance increase in disk reading time compared to the non-incremental algorithm with small updates sizes. The downside of the incremental algorithm is that it has 60% higher memory overhead compared to the non-
incremental one, caused by data structures required to enable fast IEJoin updates. I also compare the performance difference between $B$-$\Delta Inc$ and $\sigma Inc$ in Figure 4.16(b) using 30M tuples as cached input, where the only difference between them is the size of the output. $\sigma Inc$ is at most 20% slower than $B$-$\Delta Inc$ when the output difference between them is large on the 1M tuples update. However, the performance gap between $B$-$\Delta Inc$ and $\sigma Inc$ becomes negligible as the output difference gets smaller on larger update sizes.

**Incremental experiments summary.** When the update size is smaller than 50, the PMA-based incremental algorithm performs 30% better than the sort-merge incremental algorithm and three times better than the original IEJoin. As I increase the size of the updates, the batched sort-merge-based incremental algorithm becomes more efficient than the PMA-based one. For update sizes that do not exceed 25% of the cached input size, the batched sort-merge-based incremental algorithm is twice faster than the original algorithm.

### 4.7.9 Multi-node Experiments

I now evaluate my proposal in a distributed environment and by using larger datasets.

**Scalable IEJoin vs. baseline systems.** I should note that I had to run these experiments on a cluster of 6 compute nodes only due to the limit imposed by the free version of the distributed PostgreSQL system. Additionally, in these experiments, I stopped the execution of any system that exceeded 24 hours. I test the scalable IEJoin using the parallel IEJoin (with 6 threads in a single machine while enabling disk caching), and the distributed IEJoin (on 6 compute nodes). Figure 4.17 shows the results of all distributed systems I consider for queries $Q_1$ and $Q_2$. This figure shows again that both versions of my algorithm significantly outperform all baselines. It is on average more than one order of magnitude faster. In particular, I observe that only DPG-GiST could terminate before 24 hours for $Q_2$. The distributed IEJoin is
twice faster than the time required to run GiST indexing alone. Moreover, distributed IEJoin is, as expected, faster than the parallel multi-threaded version. This is because the multi-threaded version has a higher processing overhead due to resource contention. These results show the high superiority of my algorithm over all baseline systems.

**Scaling input size.** I further push the evaluation of efficiency in a distributed environment with bigger input datasets: from 100M to 500M records with large results size (Employees and Events), and from 1B to 6B records with smaller results size (Employees2 and Events2). As I now consider IEJoin only, I run this experiment on my entire 16 compute nodes cluster. Figure 4.18 shows the runtime results as well as the output sizes. I observe that IEJoin gracefully scales along with input dataset size in both scenarios. I also observe in Figure 4.18(a) that, when the output size is large, the runtime increases accordingly as it is dominated by the materialization of the results. In Figure 4.18(a) $Q_1$ is slower than $Q_2$ as its output is three orders of magnitude larger. When the output size is relatively small, both $Q_1$ and $Q_2$ scale well with increasing input size (see Figure 4.18(b)). Below, I study in more details
the impact of the output size on performance.

**Scaling dataset output size.** I test my system’s scalability in terms of the output size using two real datasets (MDC and Cloud), as shown in Figure 4.19. To have full control on this experiment, I explicitly limit the output size from 4.3M to 430M for MDC, and 20.8M to 2050M for Cloud. The figures clearly show that the output size affects runtime; the larger the output size, the longer it will take to produce them. They also show that materializing a large number of results is costly. Take
Figure 4.19: Runtime of IEJoin ($c = 10$)

Figure 4.19(a) for example, when the output size is small (i.e., 4.3M), materializing them or not will have similar performance. However, when the output size is big (i.e., 430M), materializing the results takes almost 2/3 of the entire running time.

In order to run another set of experiments with a much bigger output size, I created two variants of $Q_3$ for MDC dataset by keeping only two predicates over four (less selectivity).

Figure 4.20 shows the scalability results of these experiments with no materialization of results.

For $Q_{3a}$, IEJoin produced more than 1,000B records in less than 3,000 seconds. For $Q_{3b}$, I stopped the execution after 2 hours with more than 5,000B tuples in the temporary result. This demonstrates the good scalability of my solution.

**Speedup and scaleup.** I also test speedup and scaleup efficiency of the distributed IEJoin by using Employees2 dataset and query $Q_1$. Figure 4.21(a) shows that my algorithm has outstanding speedup thanks to the scalability optimizations. IEJoin was only 4%, 3% and 16% slower than the ideal speedup when processing 8B rows on 4, 8 and 16 workers respectively. Figure 4.21(b) shows the scaleup efficiency of IEJoin as I proportionally increase the cluster size and input size. I observe that
distributed IEJoin also has good scaleup: on 4 workers (2B rows) and 8 workers (4B rows) it was only 5% and 20% slower than the ideal scaleup. However, due to the increase in dataset size, the sorting overhead in IEJoin becomes larger. This explains why scalable IEJoin, on 16 workers with 8B rows input, is 46% slower than the ideal scaleup.

**Multi-node summary.** Similarly to the centralized case, IEJoin outperforms existing baselines by at least one order of magnitude. In particular, I observe that it
gracefully scales in terms of input (up to 6B tuples). This is because my algorithm first join the metadata, which are orders of magnitude smaller than the actual data. As a result, it shuffles only those data partitions that can potentially produce join results. Typically, IEJoin processes a small number of data partitions.
Chapter 5

Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing

In this chapter, I present Mizan as a solution to achieve dynamic load balancing in distributed graph computations. First, I analyze different graph algorithm characteristics that can contribute to an imbalanced computation in a Pregel system (Chapter 5.1). After that, I propose a dynamic vertex migration model based on runtime monitoring of vertices to optimize the end-to-end computation (Chapter 5.2). The proposed migration model is fully implemented in C++ as an optimized Pregel system (Mizan) that supports dynamic load balancing and efficient vertex migration (Chapter 5.3). Finally, I deploy Mizan on a local Linux cluster (21 machines) and evaluate its efficacy on a representative number of datasets. I also show the linear scalability of my design by running Mizan on a 1024-CPU IBM Blue Gene/P (Chapter 5.4). Mizan is not intended exclusively to empower BigDancing only; it is an independent general system that adapts to various graph algorithms and input structure without affecting the BSP programming model of Pregel.

5.1 Dynamic Behavior of Algorithms

In the Pregel (BSP) computing model, several factors can affect the runtime performance of the underlying system (shown in Figure 5.1). During a superstep, each vertex is either in an active or inactive state. In an active state, a vertex may be computing, sending messages, or processing received messages.
Figure 5.1: Factors that can affect the runtime in the Pregel framework

Naturally, vertices can experience variable execution times depending on the algorithm they implement or their degree of connectivity. A densely connected vertex, for example, will likely spend more time sending and processing incoming messages than a sparsely connected vertex. The BSP programming model, however, masks this variation by (1) overlapping communication and computation, and (2) running many vertices on the same compute node. Intuitively, as long as the workloads of vertices are roughly balanced across computing nodes, the overall computation time will be minimized.

Counter to intuition, achieving a balanced workload is not straightforward. There are two sources of imbalance: one originating from the graph structure and another from the algorithm behavior. Different applications generate various graph structures while not all user algorithms behave similarly. In both cases, vertices on some compute nodes can spend a disproportional amount of time computing, sending or processing messages. In some cases, these vertices can run out of input buffer capacity and start paging, further exacerbating the imbalance.

Existing implementations of Pregel (including Giraph \[29\], GoldenOrb \[30\],

\[\text{[...]}\]
Hama [31] focus on providing multiple alternatives to partitioning the graph data. The three common approaches to partitioning the data are hash-based, range-based, or min-cut [33]. Hash- and range-based partitioning methods divide a dataset based on a simple heuristic: to evenly distribute vertices across compute nodes, irrespective of their edge connectivity. Min-cut based partitioning, on the other hand, considers vertex connectivity and partitions the data such that it places strongly connected vertices close to each other (i.e., on the same cluster). The resulting performance of these partitioning approaches is, however, graph dependent. To demonstrate this variability, I ran a simple—and highly predictable—PageRank algorithm on different datasets (summarized in Table 5.1) using the three popular partitioning methods. Figure 5.2 shows that none of the partitioning methods consistently outperforms the rest, noting that ParMETIS [93] partitioning cannot be performed on the arabic-2005 graph due to memory limitations.

In addition to the graph structure, the running algorithm can also affect the workload balance across compute nodes. Broadly speaking, graph algorithms can be divided into two categories (based on their communication characteristics across supersteps): stationary and non-stationary.

**Stationary Graph Algorithms.** An algorithm is stationary if its active vertices send and receive the same distribution of messages across supersteps. At the end of a stationary algorithm, all active vertices become inactive (terminate) during the same superstep. Graph algorithms represented by a matrix-vector multiplication\footnote{The matrix represents the graph adjacency matrix and the vector represents the vertices’ value.} usually are stationary algorithms, including PageRank, diameter estimation and finding weakly connected components.

**Non-stationary Graph Algorithms.** A graph algorithm is non-stationary if the destination or size of its outgoing messages changes across supersteps. Such variations can create workload imbalances across supersteps. Examples of non-stationary
algorithms include distributed minimal spanning tree construction (DMST), graph queries, graph coloring, and various simulations on social network graphs (e.g., advertisement propagation).

To illustrate the differences between the two classes of algorithms, I compared the runtime behavior of PageRank (stationary) against DMST (non-stationary) when processing the same dataset (LiveJournal1) on a cluster of 21 machines. The input graph was partitioned using a hash function. Figure 5.3 shows the variability in the incoming messages per superstep for all workers. In particular, the variability can span over five orders of magnitude for non-stationary algorithms and the distribution of incoming messages across machines within a single superstep becomes highly variable. The example in Figure 5.3 shows that a single machine received more than 80% of the total messages received by all machines on supersteps 49, 52 and 55.

The remainder of the section describes four popular graph mining algorithms that I use throughout the chapter. They cover both stationary and non-stationary algorithms.
Figure 5.3: The difference between stationary and non-stationary graph algorithms with respect to the incoming messages. *Total* represents the sum across all workers and *Max* represents the maximum amount (on a single worker) across all workers.

### 5.1.1 Example Algorithms

**PageRank.** PageRank is a stationary algorithm that uses matrix-vector multiplications to calculate the eigenvalues of the graph’s adjacency matrix at each iteration. The algorithm terminates when the PageRank values of all nodes change by less than a defined error during an iteration. At each superstep, all vertices are active, where every vertex always receives messages from its in-edges and sends messages to all of its out-edges. All messages have fixed size (8 bytes) and the computation complexity on each vertex is linear to the number of messages.

**Top-K Ranks in PageRank.** It is often desirable to perform graph queries, like using PageRank to find the top $k$ vertex ranks reachable to a vertex after $y$ supersteps. In this case, PageRank runs for $x$ supersteps; at superstep $x + 1$, each vertex sends its rank to its direct *out* neighbors, receives other ranks from its *in* neighbors, and stores the highest received $k$ ranks. At supersteps $x + 2$ and $x + y$, each vertex sends the top-K ranks stored locally to its direct *out* neighbors, and again stores

---

$^2$Formally, each iteration calculates: $v^{(k+1)} = cA^tv^k + (1 - c)/|N|$, where $v^k$ is the eigenvector of iteration $k$, $c$ is the damping factor used for normalization, and $A$ is a row-normalized adjacency matrix.
the highest $k$ ranks received from its in neighbors. The message size generated by each vertex can be between $[1..k]$, which depends on the number of ranks stored the vertex. If the highest $k$ values maintained by a vertex did not change at a superstep, that vertex votes to halt and becomes inactive until it receives messages from other neighbors. The size of the message sent between two vertices varies depending on the actual ranks. I classify this algorithm as non-stationary because of the variability in message size and number of messages sent and received.

**Distributed Minimal Spanning Tree (DMST).** Implementing the *GHS* algorithm of Gallager, Humblet and Spira \cite{135} in the Pregel model requires that, given a weighted graph, every vertex classifies the state of its edges as (1) *branch* (i.e., belongs to the minimal spanning tree (MST)), (2) *rejected* (i.e., does not belong to the MST), or (3) *basic* (i.e., unclassified), in each superstep.

The algorithm starts with each vertex as a fragment, then joins fragments until there is only one fragment left (i.e., the MST). Each fragment has an ID and level. Each vertex updates its fragment ID and level by exchanging messages with its neighbors. The vertex has two states: active and inactive. There is also a list of auto-activated vertices defined by the user. At the early stage of the algorithm, active vertices send messages over the minimum weighted edges (i.e., the edge that has the lowest weight among the other in-edges and out-edges). As the algorithm progresses, more messages are sent across many edges according to the number of fragments identified during the MST computations at each vertex.

At the last superstep, each vertex knows which of its edges belong to the minimum weighted spanning tree. The computational complexity for each vertex is quadratic to the number of incoming messages. I classify this algorithm as non-stationary because the flow of messages between vertices is unpredictable and depends on the state of the edges at each vertex.

**Simulating Advertisements on Social Networks.** To simulate advertisements
on a social network graph, each vertex represents a profile of a person containing a list of his/her interests. A small set of selected vertices is identified as sources and send different advertisements to their direct neighbors at each superstep for a predefined number of supersteps. When a vertex receives an advertisement, it is either forwarded to the vertex’s neighbors (based on the vertex's interest matrix) or ignored. This algorithm has a computation complexity that depends on the count of active vertices at a superstep and whether the active vertices communicate with their neighbors or not. It is, thus, a non-stationary algorithm.

I have also implemented and evaluated a number of other—stationary and non-stationary—algorithms including diameter estimation and finding weakly connected components and they had consistent behavior with the results in this chapter.

5.2 Mizan

MIZAN is a BSP-based graph processing system that is similar to Pregel, but focuses on efficient dynamic load balancing of both computation and communication across all worker (compute) nodes. Like Pregel, MIZAN first reads and partitions the graph data across workers. The system then proceeds as a series of supersteps, each separated by a global synchronization barrier. During each superstep, each vertex processes incoming messages from the previous superstep and sends messages to neighboring vertices (which are processed in the following superstep). Unlike Pregel, MIZAN balances its workload by moving selected vertices across workers. Vertex migration is performed when all workers reach a superstep synchronization barrier to avoid violating the computation integrity, isolation and correctness of the BSP compute model. This section describes two aspects of the design of MIZAN related to vertex migration: the distributed runtime monitoring of workload characteristics and a distributed migration planner that decides which vertices to migrate. Other design aspects that are not related to vertex migration (e.g., fault tolerance) follow similar
5.2.1 Monitoring

MIZAN monitors three key metrics for each vertex and maintains a high-level summary these metrics for each worker node; summaries are broadcast to other workers at the end of every superstep. As shown in Figure 5.4, the key metrics for every vertex are (1) the number of outgoing messages to other (remote) workers, (2) total incoming messages, and (3) the response time (execution time) during the current superstep:

**Outgoing Messages.** Only outgoing messages to other vertices in remote workers are counted since the local outgoing messages are rerouted to the same worker and do not incur any actual network cost.

**Incoming Messages.** All incoming messages are monitored, those that are received from remote vertices and those locally generated. This is because queue size can affect the performance of the vertex (i.e., when its buffer capacity is exhausted, paging to disk is required).

**Response Time.** The response time for each vertex is measured. It is the time when a vertex starts processing its incoming messages until it finishes.

Figure 5.4: The statistics monitored by MIZAN

approaches to Pregel [20], Giraph [29], GraphLab [92] and GPS [96]; they are omitted for space considerations. Implementation details are described in Chapter 5.3.
5.2.2 Migration Planning

High values for any of the three metrics above may indicate poor vertex placement, which leads to workload imbalance. As with many constrained optimization problems, optimizing against three objectives is non-trivial and costly. To reduce the optimization search space, MIZAN’s migration planner finds the strongest cause of workload imbalance among the three metrics and plans the vertex migration accordingly. Since distinct supersteps do not necessary share the same workload imbalance cause, MIZAN applies the best possible migration plan to each superstep independently and achieves its optimal state incrementally.

By design, all workers create and execute the migration plan in parallel, without requiring any centralized coordination. The migration planner starts on every worker at the end of each superstep (i.e., when all workers reach the synchronization barrier), after it receives summary statistics (as described in Chapter 5.2.1) from all other workers. Additionally, the execution of the migration planner is sandwiched between a second synchronization barrier as shown in Figure 5.5, which is necessary to ensure correctness of the BSP model. MIZAN’s migration planner executes the following five steps, summarized in Figure 5.6.

1. Identify the source of imbalance.
2. Select the migration objective (i.e., optimize for outgoing messages, incoming messages, or response time).

3. Pair over-utilized workers with under-utilized ones.

4. Select vertices to migrate.

5. Migrate vertices.

**STEP 1: Identify the source of imbalance.** 
MIZAN detects imbalances across supersteps by comparing the summary statistics of all workers against a normal random distribution and flagging outliers. Specifically, at the end of a superstep, MIZAN computes the z-score for all workers. If any worker has z-score greater than \( z_{def} \), MIZAN’s migration planner flags the superstep as imbalanced. We found that \( z_{def} = 1.96 \), the commonly recommend value \[136\], allows for natural workload fluctuations across workers. We have experimented with different values of \( z_{def} \) and validated the robustness of my choice.

**STEP 2: Select the migration objective.** MIZAN compares the effect of sending messages over execution time versus the effect of processing incoming messages over execution time to find the dominating factor that affected the computation balance. That is, each worker—identically—uses the summary statistics to compute the correlation between outgoing messages and computation time, and also the correlation between incoming messages and computation time. The correlation scores are used to select the objective to optimize for: to balance outgoing messages, balance incoming messages, or balance computation time. The larger the correlation between the outgoing messages and computation time the higher the probability that outgoing messages is the dominating factor, which is also true for the incoming messages. The default objective is to balance the computation time. If the outgoing or incoming messages

3The z-score \( Wz_i = \frac{|x_i - max(x)|}{\text{standard deviation}} \), where \( x_i \) is the runtime of worker \( i \) and max\( (x) \) is the largest runtime across all workers
are highly correlated with the response time, then MIZAN chooses the objective with highest correlation score.

The correlation scores are calculated using two tests: clustering and sorting. In the clustering test $CT$, elements from the set of worker response times $S_{Time}$ and the statistics set $S_{Stat}$ (either incoming or outgoing message statistics) are divided into two categories based on their z-score\footnote{The z-score $S_{Time}^i$ and $S_{Stat}^i = \frac{|x_i - \mu_{Time}|}{\text{standard deviation}}$, where $x_i$ is the run time of worker $i$ or its statistics} positive and negative clusters. Elements from a set $S$ are categorized as positive if their z-scores are greater than some $z_{def}$. Elements categorized as negative have a z-score less than $-z_{def}$. Elements in the range $(-z_{def}, z_{def})$ are ignored. The score of the cluster test is calculated based on the total count of elements that belongs to the same worker in the positive and negative sets, which is shown by the following equation:
The sorting test \( ST \) consists of sorting both sets, \( S_{\text{Time}} \) and \( S_{\text{Stat}} \), and calculating the z-scores for all elements. Elements with a z-score in the range \( (-z_{\text{def}}, z_{\text{def}}) \) are excluded from both sorted sets. For the two sorted sets \( S_{\text{Time}} \) and \( S_{\text{Stat}} \) of size \( n \) and \( i \in [0..n-1] \), a mismatch is defined if the \( i \)-th elements from both sets are related to different workers. That is, \( S_{\text{Time}}^i \) is the \( i \)-th element from \( S_{\text{Time}} \), \( S_{\text{Stat}}^i \) is the \( i \)-th element from \( S_{\text{Stat}} \), and \( S_{\text{Time}}^i \in \text{Worker}_x \) while \( S_{\text{Stat}}^i \notin \text{Worker}_x \). A match, on the other hand, is defined if \( S_{\text{Time}}^i \in \text{Worker}_x \) and \( S_{\text{Stat}}^i \in \text{Worker}_x \). A score of 1 is given by the sorting test if:

\[
ST_{\text{score}} = \frac{\text{count}(\text{matches})}{\text{count}(\text{matches}) + \text{count}(\text{mismatches})}
\]

The correlation of sets \( S_{\text{Time}} \) and \( S_{\text{Stat}} \) are reported as strongly correlated if \( (CT + ST)/2 \geq 1 \), reported as weakly correlated if \( 1 > (CT + ST)/2 \geq 0.5 \) and reported as uncorrelated otherwise.

**STEP 3: Pair over-utilized workers with under-utilized ones.** Each over-utilized worker that needs to migrate vertices out is paired with a single unique underutilized worker. In other words, an underutilized worker at most is paired with a single over-utilized worker. Only workers with a z-score greater than \( z_{\text{def}} \) are marked as over-utilized workers. While complex pairings are possible, we choose a design that is efficient to execute, especially since the exact number of vertices that different workers plan to migrate is not globally known. Similar to the previous two steps in the migration plan, this step is executed by each worker without explicit global synchronization. Using the summary statistics for the chosen migration objective (in Step 2), each worker creates an ordered list of all workers. For example, if the objective is to balance outgoing messages, then the list will order workers from highest to lowest
outgoing messages. The resulting list, thus, places over-utilized workers at the top and least utilized workers at the bottom. The pairing function then successively matches workers from opposite ends of the ordered list. As depicted in Figure 5.7, if the list contains \( n \) elements (one for each worker), then the worker at position \( i \) is paired with the worker at position \( n - i \). In cases where a worker does not have memory to receive any vertices or has no match with an over-utilized one, the worker is marked unavailable in the list.

**STEP 4: Select vertices to migrate.** The number of vertices to be selected from an over-utilized worker depends on the difference of the selected migration objective statistics with its paired worker. Assume that \( w_x \) is a worker that needs to migrate out a number of vertices, and is paired with the receiver, \( w_y \). The load that should be migrated to the underutilized worker is defined as \( \Delta_{xy} \), which equals to half the difference in statistics of the migration objective between the two workers. The selection criteria of the vertices depends on the distribution of the statistics of the migration objective, where the statistics of each vertex is compared against a normal distribution. A vertex is selected if it is an outlier\(^5\) (i.e., if its \( V_{z_{stat}}^i > z_{def} \)). For example, if the migrating objective is to balance the number of remote outgoing messages, vertices with large remote outgoing messages are selected to migrate to the underutilized worker. The sum of the statistics of the selected vertices is denoted by \( \sum V_{stat} \) which should minimize \( |\Delta_{xy} - \sum V_{stat}| \) to ensure the balance between \( w_x \) and \( w_y \) in the next superstep. If there not enough outlier vertices are found, a random set of vertices are selected to minimize \( |\Delta_{xy} - \sum V_{stat}| \). This step reduces migration communication and processing cost through minimizing the number of vertices required to migrate from over-utilized workers to underutilized ones. The migrating vertex count depends on the workload distribution of vertices, where the maximum number of migrating

\(^5\)Requires more execution time compared to the average vertex at the same worker

\(^6\)The z-score \( V_{z_{stat}}^i = \frac{|x_i - \text{avg}(x)|}{\text{standard deviation}} \), where \( x_i \) is the statistics of the migration objective of vertex \( i \) is greater than the \( z_{def} \).
vertices—in the worst case scenario—is at most half the number of vertices in the over-utilized worker.

**STEP 5: Migrate vertices.** After the vertex selection process, the migrating workers start sending the selected vertices while other workers wait at the migration barrier. A migrating worker starts sending the selected set of vertices to its unique target worker, where each vertex is encoded into a stream that includes the vertex ID, state, edge information and the received messages it will process. Once a vertex stream is successfully sent, the sending worker deletes the sent vertices so that it does not run them in the next superstep. The receiving worker, on the other hand, receives vertices (together with their messages) and prepares to run them in the next superstep. The next superstep is started once all workers finish migrating vertices and reach the migration barrier. The complexity of the migration process is directly related to the size of vertices being migrated.

### 5.3 Implementation

**MIZAN** consists of four modules that compose the main building blocks toward the graph processing system proposed in this thesis, shown in Figure 5.8: the *BSP Graph Processor*, *Storage Manager*, *Communicator*, and *Migration Planner*. The BSP Graph Processor implements the Pregel APIs, consisting primarily of the *Compute* class, and the *SendMessageTo*, *GetOutEdgeIterator* and *getValue* methods. The BSP Processor operates on the data structures of the graph and executes the user’s algorithm.
It also performs barrier synchronization with other workers at the end of each superstep. The *Storage Manager* module maintains access atomicity and correctness of the graph’s data, and maintains the data structures for message queues. Graph data can be read and written to either HDFS or local disks, depending on how MIZAN is deployed. HDFS is also used in MIZAN as a communication channel to BIGDANSING for exchanging violation graphs. The Communicator module uses MPI to enable communication between workers; it also maintains distributed vertex ownership information. Finally, the *Migration Planner* transparently maintains the dynamic workload balance across superstep barriers.

MIZAN allows the user’s code to manipulate the graph connectivity by adding and removing vertices and edges at any superstep. It also guarantees that all graph mutation commands issued at superstep $x$ are executed at the end of the same superstep and before the BSP barrier, which is illustrated in Figure 5.5. Therefore, vertex migrations performed by MIZAN do not conflict with the user’s graph mutations and MIZAN always considers the most recent graph structure for migration planning.

When implementing MIZAN, I wanted to avoid having a centralized controller. Overall, the BSP (Pregel) model naturally lends itself to a decentralized implementation. There were, however, three key challenges in implementing a distributed control plane that supports fine-grained vertex migration. The first challenge was in maintaining vertex ownership so that vertices can be freely migrated across workers. This is different from existing approaches (e.g., Pregel, Giraph, and GoldenOrb), which operate on a much coarser granularity (clusters of vertices) to enable scalability. The second challenge was in allowing fast updates to the vertex ownership information as vertices get migrated. The third challenge was in minimizing the cost of migrating vertices with large data structures. In this section, I discuss the implementation details around these three challenges, which allow MIZAN to achieve its effectiveness and scalability.
5.3.1 Vertex Ownership

With huge datasets — graphs at the scale of billion vertices — MIZAN workers cannot maintain the management information for all vertices in the graph. Management information includes the collected statistics for each vertex (described in Chapter 5.2.1) and the location (ownership) of each vertex. While per-vertex monitoring statistics are only used locally by the worker, vertex ownership information is needed by all workers. When vertices send messages, workers need to know the destination address for each message. With frequent vertex migration, updating the location of the vertices across all workers can easily create a communication bottleneck.

To overcome this challenge, I use a distributed hash table (DHT) to implement a distributed lookup service that scales to thousands of workers. The DHT implementation allows MIZAN to distribute the overhead of looking up and updating vertex location across all workers. The DHT stores a set of (key,value) pairs, where the key represents a vertex ID and the value represents its current physical location. Each vertex is assigned a home worker. The role of the home worker is to maintain the current location of the vertex. A vertex can physically exist in any worker, including its home worker. The DHT uses a globally defined hash function that maps the keys to their associated home workers, such that $\text{home\_worker} = \text{location\_hash(key)}$. 

---

Figure 5.8: Architecture of MIZAN in light of the proposed graph processing system
During a superstep, when a (source) vertex sends a message to another (target) vertex, the message is passed to the Communicator. If the target vertex is located on the same worker, it is rerouted back to the appropriate queue. Otherwise, the source worker uses the location_hash function to locate and query the home worker for the target vertex. The home worker responds back with the actual physical location of target vertex. The source worker finally sends the queued message to the current worker for the target vertex. It also caches the physical location to minimize future lookups.

5.3.2 DHT Updates After Vertex Migration

Figure 5.9 depicts the vertex migration process. When a vertex \( v \) migrates between two workers, the receiving worker sends the new location of \( v \) to the home worker of \( v \). The home worker, in turn, sends an update message to all workers that have previously asked for—and, thus, potentially cached—the location of \( v \). Since MIZAN migrates vertices in the barrier between two supersteps, all workers that have cached the location of the migrating vertex will receive the updated physical location from
the home worker before the start of the new superstep.

If for any reason a worker did not receive the updated location, the messages will be sent to the last known physical location of \( v \). The receiving worker, which no longer owns the vertex, will simply buffer the incorrectly routed messages, ask for the new location for \( v \), and reroute the messages to the correct worker.

### 5.3.3 Migrating Vertices with Large Message Size

Migrating a vertex to another worker requires moving its queued messages and its entire state (which includes its ID, value, and neighbors). Especially when processing large graphs, a vertex can have a significant number of queued messages, which are costly to migrate. To minimize the cost, Mizan migrates large vertices using a *delayed migration* process that spreads the migration over two supersteps. Instead of physically moving the vertex with its large message queue, delayed migration only moves the vertex’s information and the ownership to the new worker. Assuming \( w_{\text{old}} \) is the old owner and \( w_{\text{new}} \) is the new one, \( w_{\text{old}} \) continues to process the migrating vertex \( v \) in the next superstep, \( SS_{t+1} \). \( w_{\text{new}} \) receives the messages for \( v \), which will
be processed at the following superstep, $SS_{t+2}$. At the end of $SS_{t+1}$, $w_{old}$ sends the new value of $v$, calculated at $SS_{t+1}$, to $w_{new}$ and completes the delayed migration. Note that migration planning is disabled for superstep $SS_{t+1}$ after applying delayed migration to ensure the consistency of the migration plans.

An example is shown in Figure 5.10 where Vertex 7 is migrated using delayed migration. As Vertex 7 migrates, the ownership of Vertex 7 is moved to Worker new and messages sent to Vertex 7 at $SS_{t+1}$ are addressed to Worker new. At the barrier of $SS_{t+1}$, Worker old sends the edge information and state of $v$ to Worker new and starts $SS_{t+2}$ with $v$ fully migrated to Worker new.

With delayed migration, the consistency of computation is maintained without introducing additional network cost for vertices with large message queues. Since delayed migration uses two supersteps to complete, Mizan may need more steps before it converges on a balanced state.

### 5.4 Evaluation

I implemented Mizan using C++ and MPI and compared it against Giraph [29], a Pregel clone implemented in Java on top of Hadoop. I ran my experiments on a local cluster of 21 machines equipped with a mix of i5 and i7 processors with 16GB RAM on each machine. I also used an IBM Blue Gene/P supercomputer with 1024 PowerPC-450 CPUs, each with 4 cores at 850MHz and 4GB RAM. I downloaded

| $G(N, E)$ | $|N|$       | $|E|$       |
|----------|------------|------------|
| kg1      | 1,048,576  | 5,360,368  |
| kg4m68m  | 4,194,304  | 68,671,566 |
| web-Google | 875,713 | 5,105,039  |
| LiveJournal1 | 4,847,571 | 68,993,773 |
| hollywood-2011 | 2,180,759 | 228,985,632 |
| arabic-2005 | 22,744,080 | 639,999,458 |

Table 5.1: Datasets—$N$, $E$ denote nodes and edges, respectively. Graphs with prefix kg are synthetic.
publicly available datasets from the Stanford Network Analysis Project[^1] and from The Laboratory for Web Algorithmics (LAW) [138, 139]. I also generated synthetic datasets using the Kronecker [140] generator that models the structure of real life networks. The details are shown in Table 5.1.

To better isolate the effects of dynamic migration on system performance, I implemented three variations of MIZAN: Static, Work Stealing (WS), and MIZAN. Static MIZAN disables any dynamic migration and uses either hash-based, range-based or min-cuts graph pre-partitioning (rendering it similar to Giraph). Work Stealing (WS) MIZAN is my attempt to emulate Pregel’s coarse-grained dynamic load balancing behavior (described in [20]). Finally, MIZAN is my framework that supports dynamic migration as described in Chapters 5.2 and 5.3.

I evaluated MIZAN across three dimensions: partitioning scheme, graph structure, and variable runtime behaviors using stationary and non-stationary algorithms.

I partition the datasets using three schemes: hash-based, range-based, and METIS partitioning. I run my experiments against both social and random graphs. Finally, I experimented with the algorithms described in Chapter 5.1.1.

### 5.4.1 Giraph vs. MIZAN

I picked Giraph for its popularity as an open source Pregel framework with broad adoption. I only compared Giraph to Static MIZAN, which allows me to evaluate the base (non-dynamic) implementation. MIZAN’s dynamic migration is evaluated later in this section. To further equalize Giraph and the Static MIZAN, I disabled the fault tolerance feature of Giraph to eliminate any additional overhead resulting from frequent snapshots during runtime. In this section, I report the results using PageRank (a stationary and well-balanced algorithm) using different graph structures.

In Figure 5.11 I ran both frameworks on social network and random graphs. As

[^1]: [http://snap.stanford.edu](http://snap.stanford.edu)
Figure 5.11: Comparing Static Mizan vs. Giraph using PageRank on social network and random graphs

shown, Static Mizan outperforms Giraph by a large margin, up to four times faster than Giraph in kg4m68, which contains around 70M edges. I also compared both systems when increasing the number of nodes in random structure graphs. As shown in Figure 5.12, Static Mizan consistently outperforms Giraph in all datasets and reaches up to three times faster with 16 million vertexes. While the execution time of both frameworks increases linearly with graph size, the rate of increase—slope of the graph—for Giraph (0.318) is steeper than Mizan (0.09), indicating that Mizan also achieves better scalability.

The experiments in Figures 5.11 and 5.12 show that Giraph’s implementation is inefficient. It is a non-trivial task to discover the source of inefficiency in Giraph since it is tightly coupled with Hadoop. I suspect that part of the inefficiency is due to the initialization cost of the Hadoop jobs and the high overhead of communication. Other factors, like internal data structure choice and memory footprint, might also play a role in this inefficiency.
Figure 5.12: Comparing **MIZAN** vs. Giraph using PageRank on regular random graphs, the graphs are uniformly distributed with each has around 17M edge.

Figure 5.13: Comparing **Static Mizan** and Work Stealing (Pregel clone) vs. **MIZAN** using PageRank on a social graph (**LiveJournal**). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph.

### 5.4.2 Effectiveness of Dynamic Vertex Migration

Given the large performance difference between **Static MIZAN** and Giraph, I exclude Giraph from further experiments and focus on isolating the effects of dynamic migration on the overall performance of the system.
Figure 5.14: Comparing MIZAN’s migration cost with the algorithm’ runtime at each superstep using PageRank on a social graph (LiveJournal1) starting with range based partitioning. The points on the superstep’s runtime trend represents MIZAN’s migration objective at that specific superstep.

Figure 5.13 shows the results for running PageRank on a social graph with various partitioning schemes. I notice that both hash-based and METIS partitioning achieve a balanced workload for PageRank such that dynamic migration did not improve the results. In comparison, range-based partitioning resulted in poor graph partitioning. In this case, I observe MIZAN (with dynamic partitioning) was able to reduce execution time by approximately 40% when compared to the static version. I have also evaluated the diameter estimation algorithm, which behaves the same as PageRank, but exchanges larger size messages. MIZAN exhibited similar behavior with diameter estimation; the results are omitted for space considerations.

Figure 5.14 shows how MIZAN’s dynamic migration was able to optimize running PageRank starting with range-based partitioning. The figure shows that MIZAN’s migration reduced both the variance in workers’ runtime and the superstep’s runtime. The same figure also shows that MIZAN’s migration alternated the optimization objective between the number of outgoing messages and vertex response time, illustrated as points on the superstep’s runtime trend.

By looking at both Figures 5.14 and 5.15, I observe that the convergence of
Figure 5.15: Comparing both the total migrated vertices and the maximum migrated vertices by a single worker for PageRank on LiveJournal starting with range-based partitioning. The migration cost at each superstep is also shown.

MIZAN’s dynamic migration is correlated with the algorithm’s runtime reduction. For the PageRank algorithm with range-based partitioning, MIZAN requires 13 supersteps to reach an acceptable balanced workload. Since PageRank is a stationary algorithm, MIZAN’s migration converged quickly; I expect that it would require more supersteps to converge on other algorithms and datasets. In general, MIZAN requires multiple supersteps before it balances the workload. This also explains why running MIZAN with range-based partitioning is less efficient than running MIZAN with METIS or hash-based partitioning.

As described in Chapter 5.1.1, Top-K PageRank adds variability in the communication among the graph nodes. As shown in Figure 5.16, such variability in the messages exchanged leads to minor variation in both hash-based and METIS execution times. Similarly, in range partitioning, MIZAN had better performance than the static version. The slight performance improvement arises from the fact that the base algorithm (PageRank) dominates the execution time. If a large number of queries are performed, the improvements will be more significant.

To study the effect of algorithms with highly variable messaging patterns, I evaluated MIZAN using two algorithms: DMST and advertisement propagation simulation.
In both cases, I used METIS to pre-partition the graph data. METIS partitioning groups the strongly connected subgraphs into clusters, thus minimizing the global communication among each cluster.

Figure 5.16: Comparing static MIZAN and Work Stealing (Pregel clone) vs. MIZAN using Top-K PageRanks on a social graph (LiveJournal1). The shaded part of each column represents the algorithm runtime while unshaded parts represents the initial partitioning cost of the input graph.

In DMST, as discussed in Chapter 5.1.1, computation complexity increases with vertex connectivity degree. Because of the quadratic complexity of computation as a function of connectivity degree, some workers will suffer from extensive workload while others will have light workload. Such imbalance in the workload leads to the results shown in Figure 5.17. MIZAN was able to reduce the imbalance in the workload, resulting in a significant drop in execution time (two orders of magnitude improvement). Even when using my version of Pregel’s load balancing approach (called work stealing), MIZAN is roughly eight times faster.

Similar to DMST, the behavior of the advertising propagation simulation algorithm varies across supersteps. In this algorithm, a dynamic set of graph vertices communicates heavily with each other, while others have little or no communication. In every superstep, the communication behavior differs depending on the state of the
vertex. Therefore, it creates an imbalance across the workers for every superstep. Even METIS partitioning in such case is ill-suited since workers’ load dynamically changes at runtime. As shown in Figure 5.17, similar to DMST, MIZAN is able to reduce such an imbalance, resulting in approximately 200% speedup when compared to the work stealing and static versions.

In Figures 5.18 and 5.19 I show the behavior of MIZAN’s dynamic migration with a non-stationary workload with bursts using advertising propagation algorithm with different user parameters. Due to the variability of the implemented algorithm, MIZAN’s migration planner was not able to converge to a balanced state. Instead, MIZAN predicted the runtime imbalance caused by workload bursts (such as on super-steps 10 and 19) and migrated vertices ahead of time to minimize the effect of bursts. In this experiment, MIZAN was able to reduce the workload around 20% compared to the static version and 30% compared to the Work Stealing version.

Figure 5.17: Comparing Work stealing (Pregel clone) vs. MIZAN using DMST and advertisement propagation Simulation on a metis partitioned social graph (LiveJournal11)
Figure 5.18: Comparing both the total migrated vertices and the maximum migrated vertices by a single worker for advertisement propagation on LiveJournal1 starting with METIS partitioning. The migration cost at each superstep is also shown.

Figure 5.19: Comparing static Mizan and Work Stealing (Pregel clone) vs. Mizan using advertisement propagation on a social graph (LiveJournal1) starting with METIS partitioning. The points on the Mizan’s trend represents Mizan’s migration objective at that specific superstep.

5.4.3 Overhead of Vertex Migration

To analyze migration cost, I measured the time for various performance metrics of Mizan. I used the PageRank algorithm with a range-based partitioning, and advertisement propagation simulation with METIS partitioning on the Live-Journal1 dataset using 21 workers. I chose range-based partitioning for the stationary PageRank algorithm as it provides the worst data distribution according to previous exper-
Table 5.2: Overhead of MIZAN’s migration process when compared to the total runtime using range partitioning with PageRank, and METIS partitioning with advertisement propagation Simulation on LiveJournal1 graph

Table 5.2 reports the average cost of migrating as 70.5 µs per vertex. In the LiveJournal1 dataset, MIZAN paid at most 9% penalty of the total runtime to balance the workload of both PageRank and advertisement propagation by transferring over 1M vertices with the former algorithm and 7M vertices with the latter. As shown earlier in Figure 5.13, PageRank resulted in a 40% saving in computation time when compared to Static MIZAN. Moreover, Figures 5.14 and 5.15 compare the algorithm runtime and the migration cost at each superstep, the migration cost is at most 13% (at superstep 2) and on average 6% for all supersteps that included a migration phase. Compared to PageRank, Figures 5.18 and 5.19 compare the algorithm runtime and the migration cost at each superstep for advertisement propagation. Figure 5.18 shows that only a small number of vertices were active during execution—between supersteps 1 and 7—before bursting at superstep 10, which explains the huge cost difference between the algorithm runtime and the migration overhead shown in Figure 5.19. Regardless the huge migration costs, MIZAN was able to reduce the end-to-end computation runtime around 20% compared to Static MIZAN.

5.4.4 Scalability of Mizan

I tested the scalability of MIZAN on the Linux cluster as shown in Table 5.3. I used two compute nodes as my base reference as a single node was too small when running
Table 5.3: Scalability of MIZAN on a 16 machines Linux Cluster (hollywood-2011 dataset), and an the IBM Blue Gene/P supercomputer (arabic-2005 dataset).

<table>
<thead>
<tr>
<th>Processors</th>
<th>Runtime (m)</th>
<th>Processors</th>
<th>Runtime (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>154</td>
<td>64</td>
<td>144.7</td>
</tr>
<tr>
<td>4</td>
<td>79.1</td>
<td>128</td>
<td>74.6</td>
</tr>
<tr>
<td>8</td>
<td>40.4</td>
<td>256</td>
<td>37.9</td>
</tr>
<tr>
<td>16</td>
<td>21.5</td>
<td>512</td>
<td>21.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1024</td>
<td>17.5</td>
</tr>
</tbody>
</table>

Figure 5.20: Speedup on a 16 machines Linux Cluster using PageRank on the hollywood-2011 dataset.

I was interested in performing large scale-out experiments, well beyond what can be achieved on public clouds. Since MIZAN’s was written in C++ and uses MPI for message passing, it was easily ported to IBM’s Blue Gene/P supercomputer. Once ported, I natively ran MIZAN on 1024 Blue Gene/P compute nodes. The results are shown in Table 5.3. I ran the PageRank algorithm using a huge graph (arabic-2005) that contains 639M edges. As shown in Figure 5.21, MIZAN scales linearly from 64 to 512 compute nodes then starts to flatten out as I increase to 1024 compute nodes.
Figure 5.21: Speedup on the IBM Blue Gene/P supercomputer using PageRank on the arabic-2005 dataset.

The flattening was expected since with an increased number of cores, compute nodes will spend more time communicating than computing. I expect that as I continue to increase the number of CPUs, most of the time will be spent communicating (which effectively breaks the BSP model of overlapping communication and computation).
Chapter 6

Concluding Remarks

In this chapter, I summarize my contribution in this dissertation and outline prospect future work directions.

6.1 Summary of Contributions

Data cleansing is a big data problem in both research and industry, where the non-automated efforts required to fix dirty data slow down their advancements. A 2016 survey by forbes [3] indicates that data scientists are spending 60% of their time cleaning and organizing data. When processing even larger datasets, handling sophisticated mechanism to discover errors or managing large arbitrary errors, the overhead of data cleaning may reach up to 98% of the data scientists’ time. To solve these problems, I proposed in this dissertation techniques and optimizations to scale error detection and repair.

First, I developed BigDansing; it is a system for fast and scalable big data cleansing. It enables a transparent scaling of error detection through a set of user-friendly logical operators. In other words, users may define their cleaning jobs in BigDansing without worrying about their efficiency. BigDansing takes these logical operators as input and transforms them into a physical execution plan that runs on top of general-purpose scalable systems. To maximize the effectiveness of the execution plans, BigDansing extensively optimizes them by using rule-based transformations using a set of wrappers and enhancers. When errors are detected, they
are transformed into a violation graph (or a violation hypergraph) and then a distributed graph processing system is used to efficiently split independent subgraphs. Each subgraph is an input for a unique instance of the repair algorithm, where BigDansing runs in parallel the multiple instances of the repair algorithm to increase the repair efficiency. My experiments demonstrated the superiority of BigDansing over baseline systems, for different rules on both real and synthetic datasets, with up to two order of magnitudes improvement in execution time. Moreover, BigDansing is scalable, i.e., it can detect violation on 300GB data (1907M rows) and produce 1.2TB (13 billion) violations in a few hours.

Second, I proposed IEJoin as a novel algorithm designed to efficiently handle sophisticated error discovery approaches that require inequality joins. It relies on auxiliary data structures to exploit data locality in inequality join computations with a small memory footprint. As a result, IEJoin achieves orders of magnitude computation speedup in comparison to traditional approaches. I also introduced selectivity estimation for this algorithm to support multi-predicate and multi-way inequality join queries. Moreover, I devised incremental versions of IEJoin to deal with continuous queries on changing data. I also successfully implemented IEJoin in PostgreSQL and Spark SQL as a new join operator, and in BigDansing as a new enhancer that replaces OCJoin. My experiments demonstrated that IEJoin is superior to baseline systems: it is 1.5 to 3 orders of magnitude faster than commercial and open-source centralized databases, and at least 2 orders of magnitude faster than the original Spark SQL and OCJoin. While the algorithm does not break the theoretical quadratic time bound, my experiments show performance results that are proportional to the size of the output.

Finally, I presented Mizan, a Pregel system that uses fine-grained vertex migration to load balance computation and communication across supersteps. Using distributed measurements of the performance characteristics of all vertices, Mizan
identifies the cause of workload imbalance and constructs a vertex migration plan without requiring centralized coordination. As a result, Mizan is able to optimize graph computations regardless of the structure of the violation graph which enables it to process BigDansing’s violation graph efficiently. Using a representative number of datasets and algorithms, I have showed both the efficacy and robustness of my design against varying workload conditions. I have also showed the linear scalability of Mizan, scaling to 1024 CPUs. Mizan is a general graph processing system; its optimizations are independent of the graph algorithm which makes it a suitable solution other applications.

6.2 Future Research Directions

There are several future directions for BigDansing. The first direction relates to abstracting the repairing process through logical operators, similar to violation detection. This abstraction would allow a better scalability to the repair algorithms in comparison to BigDansing’s current approach. However, abstracting the repairs is a very challenging problem because most of the existing repair algorithms use different heuristics to find optimal repairs. As a result, it becomes very difficult to find common operators for unrelated repair algorithms. Moreover, the impact of such direction might be small because data scientists are keener to use directed or interactive error repairs rather than depending on automated repair algorithms. Another direction is to exploit opportunities for multiple data quality rule optimizations, where execution plans of different repair jobs on the same dataset are merged into a single enhanced plan. This optimization would reduce the memory footprint of BigDansing, by sharing the output of logical operators, and would improve the parallelism of violation detection even further. BigDansing may also utilize the RDD caching in Spark to store the result of specific execution operators, based on the history of previous error detections, to improve their performance of future cleaning job.
A more promising direction is to derive more usable graph algorithms for BIG-DANSING to improve the repair performance while utilizing MIZAN to process such algorithms efficiently. In other words, one can implement the minimum vertex cover (MVC) algorithm on MIZAN to automatically provide a parallel version of the hypergraph repair algorithm [6,9] for BIGDANSING. Moreover, one can also use transitive reduction algorithm to compress the connected components and stream them to the repair function without losing the violations. This is useful because it also allows a faster transfer of connected components from MIZAN to BIGDANSING while avoiding overloading the memory of single machines with very large connected components.
REFERENCES


[56] “Shark (Hive on Spark),” https://github.com/amplab/shark


[107] “Alluxio: Open Source Memory Speed Virtual Distributed Storage.”


[120] Harbi, Razen and Abdelaziz, Ibrahim and Kalnis, Panos and Mamoulis, Nikos and Ebrahim, Yasser and Sahli, Majed, “Accelerating SPARQL Queries by Ex-


A Bushy Plan Example

I show a bushy plan example in Figure A.1 based on the following two tables and DC rules from [6]:

- Table Global (G): GID, FN, LN, Role, City, AC, ST, SAL
- Table Local (L): LID, FN, LN, RNK, DO, Y, City, MID, SAL

- \((c1)\): \(\forall t_1, t_2 \in G, \neg(t_1.\text{City} = t_2.\text{City} \land t_1.\text{ST} \neq t_2.\text{ST})\)
- \((c2)\): \(\forall t_1, t_2 \in G, \neg(t_1.\text{Role} = t_2.\text{Role} \land t_1.\text{City} = "NYC" \land t_2.\text{City} \neq "NYC" \land t_2.\text{SAL} > t_1.\text{SAL})\)
- \((c3)\): \(\forall t_1, t_2 \in L, t_3 \in G, \neg(t_1.\text{LID} \neq t_2.\text{LID} \land t_1.\text{LID} = t_2.\text{MID} \land t_1.\text{FN} \approx t_3.\text{FN} \land t_1.\text{LN} \approx t_3.\text{LN} \land t_1.\text{City} = t_3.\text{City} \land t_3.\text{Role} \neq "M")\)

The plan starts with applying the Scope operator. Instead of calling Scope for each rule, I only invoke Scope for each relation. Next I apply the Block operator as follows: block on “City” for c1, on “Role” for c2, and on “LID” and “MID” for c3. Thereafter, for c1 and c2, I proceed to iterate candidate tuples with violations (Iterate) and feed them to Detect and GenFix operators respectively. For c3, I iterate over all employees who are managers combine them with data units from the global table G and then finally feed them to the Detect and GenFix operators.
The key thing to note in the above bushy data cleaning plan is that while each rule has its own Detect/GenFix operator, the plan shares many of the other operators in order to reduce: (1) the number of times data is read from the base relations, and (2) the number of duplicate data units generated and processed in the dataflow.

B Spark Code for $\phi_F$ in Example 1

I show in Listing B.2 an example of the physical translation of the logical plan for rule $\phi_F$ in Spark. Lines 3 to 7 in Listing B.2 handles the processing of the UDF operators in Listings 3.1, 3.2, 3.3, 3.4 and 3.5. Lines 8 to 16 implement the repair algorithm in Chapter 3.4.2. The repair algorithm invokes Listing B.1 in line 9 to build the graph of possible fixes and finds the graph's connected components through
GraphX. Finally, lines 17 to 19 in Listing B.2 applies the selected candidate fixes to the input dataset. Note that the Spark plan translation in Listing B.2 is specific to rule $\phi_F$, where lines 3 to 16 changes according the the input rule or UDF. The repair algorithm in Listing B.2 seems far complex than the rule engine. However, as the number of violations in a dataset does not usually exceeds 10% of the input dataset size, the rule engine dominates the execution runtime because it has more data to process compared to the repair algorithm.

```scala
class RDDGraphBuilder(var edgeRDD: RDD[Edge[Fix]]) {
  def buildGraphRDD: JavaRDD[EdgeTriplet[VertexId, Fix]] = {
    var g: Graph[Integer, Fix] = Graph.fromEdges(edgeRDD, 0)
    new JavaRDD[EdgeTriplet[VertexId, Fix]](g.connectedComponents().triplets)
  }
}
```

Listing B.1: Scala code for GraphX to find connected components of fixes
//Input
1 JavaPairRDD<LongWritable, Text> inputData = sc.hadoopFile(InputPath,
    org.apache.hadoop.mapred.TextInputFormat.class,
    LongWritable.class, Text.class, minPartitions);
2 JavaRDD<Tuple> tupleData = tmpLogData.map(new StringToTuples());
//------Rule Engine------
3 JavaPairRDD<Tuple> scopeData = tupleData.map(new fdScope());
4 JavaRDD<Tuple> blockData = scopedData.groupByKey(new fdBlock());
5 JavaRDD<TuplePair> iterateData = blockingData.map(new fdIterate());
6 JavaRDD<Violation> detectData = iterateData.map(new fdDetect());
7 JavaRDD<Fix> genFixData = detectData.map(new fdGenFix());
//------Repair Algorithm------
8 JavaRDD<Edge<Fix>> edges = genFixData.map(new extractEdges());
9 JavaRDD<EdgeTriplet<Object, Fix>> ccRDD = new RDDGraphBuilder(edges.rdd()).buildGraphRDD();
10 JavaPairRDD<Integer, Fix> groupedFix = ccRDD.mapToPair(new extractCC());
11 JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>> stringFixUniqueKeys = groupedFix.flatMapToPair(new extractStringFixUniqueKey());
12 JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>> countStringFixes = stringFixUniqueKeys.reduceByKey(new countFixes());
13 JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>> newUniqueKeysStringFix = countStringFixes.mapToPair(new extractReducedCellValuesKey());
14 JavaPairRDD<Integer, Tuple4<Integer, Integer, String, String>> reducedStringFixes = newUniqueKeysStringFix.reduceByKey(new reduceStringFixes());
15 JavaPairRDD<Integer, Fix> uniqueKeysFix = groupedFix.flatMapToPair(new extractFixUniqueKey());
16 JavaRDD<Fix> candidateFixes candidateFixes = uniqueKeysFix.join(reducedStringFixes).values().flatMap(new getFixValues());
//------Apply results to input------
17 JavaPairRDD<Long, Iterable<Fix>> fixRDD = candidateFixes.keyBy(new getFixTupleID().groupByKey());
18 JavaPairRDD<Long, Tuple> dataRDD = tupleData.keyBy(new getTupleID());
19 JavaRDD<Tuple> newtupleData = dataRDD.leftOuterJoin(fixRDD).map(new ApplyFixes());

Listing B.2: Spark code for rule $\phi_F$
C Papers Published, Submitted and Under Preparation

• Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, Panos Kalnis, “A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data”, Under Review in PVLDB.