QuERy: A Framework for Integrating Entity Resolution with Query Processing

Hotham Altwaijry
Computer Science Dept.
UC Irvine

Sharad Mehrotra
Computer Science Dept.
UC Irvine

Dmitri V. Kalashnikov
AT&T Labs Research

ABSTRACT

This paper explores an analysis-aware data cleaning architecture for a large class of SPJ SQL queries. In particular, we propose QuERy, a novel framework for integrating entity resolution (ER) with query processing. The aim of QuERy is to correctly and efficiently answer complex queries issued on top of dirty data. The comprehensive empirical evaluation of the proposed solution demonstrates its significant advantage in terms of efficiency over the traditional techniques for the given problem settings.

1. INTRODUCTION

This paper addresses the problem of *analysis-aware* data cleaning, wherein the needs of the analysis task dictates which parts of the data should be cleaned. Analysis-aware cleaning is emerging as a new paradigm for data cleaning to support today's increasing demand for (near) real-time analytical applications of big data. Modern enterprises have access to potentially limitless data sources, e.g., web data repositories, social media posts, clickstream data from web portals, etc. Analysts/users usually wish to integrate one or more such data sources (possibly with their own data) to perform joint analysis and decision making. For example, a small store owner may discover an online source (e.g., a web table) containing Amazon's product pricing and may wish to compare that pricing with her own pricing.

Several systems have been developed to empower analysts to dynamically discover and merge data sources. For instance, Microsoft Power Query provides features to dynamically find, combine, visualize, share, and query data across a wide variety of online and offline sources. Another example is Trifacta [2], a data transformation platform that employs a predictive interaction framework [15] to enable users to transform raw data into structured formats. However, to the best of our knowledge, such systems have not yet incorporated data cleaning mechanisms.

Acknowledgements. This work is part of NSF supported project *Sherlock* @ UCI (http://sherlock.ics.uci.edu/). It was supported by NSF grants 1059436, 1527536, 1118114, and 1545071. Hotham Altwaijry was supported by KACST's Graduate Studies Scholarship.

Proceedings of the VLDB Endowment, Vol. 9, No. 3

As a result of merging data from a variety of sources, a given real-world object may often have multiple representations, resulting in data quality challenges. In this paper, we focus on the *Entity Resolution* (ER) challenge [6,10], the task of which is to discover duplicate entities that refer to the same real-world object and then to group them into a single cluster that uniquely represents that object.

Traditionally, entity resolution, and data cleaning in general, is performed in the context of data warehousing as an offline preprocessing step prior to making data available to analysis – an approach that works well under standard settings. Such an offline strategy, however, is not viable in emerging applications that deal with big data analysis. First, the need for (near) real-time analysis requires modern applications to execute up-to-the-minute analytical tasks, making it impossible for those applications to use time-consuming standard back-end cleaning technologies. Another reason is that in the data analysis scenarios that motivate our work, an analyst/user may discover and analyze data as part of a single integrated step. In this case, the system will know "what to clean" only at analysis time (while the user is waiting to analyze the data). Last, given the volume and the velocity of big data, it is often infeasible to expect that one can fully collect or clean data in its entirety.

Recent work on analysis-aware data cleaning seeks to overcome the limitations of traditional offline data cleaning techniques [4,8,17,22,24]. While such solutions address analysis-aware data cleaning, they are limited to only simple queries (viz., mention, selection, and/or numerical aggregation queries) executed on top of dirty data. Data analysis, however, often requires a significantly more complex type of queries requiring SQL-style joins. For instance, a user interested in comparative shopping may wish to find cellphones that are listed on two distinct data sources: *Best Buy* and *Walmart* to compare their ratings and reviews. Clearly, the query that corresponds to the user's interest will require joining Best Buy's and Walmart's cellphone-listings. In contrast to our work, the previous approaches cannot exploit the semantics of such a join predicate to reduce cleaning.

Specifically, this paper explores the problem of analysis-aware data cleaning for the general case where queries can be complex SQL-style selections and joins spanning single/multiple dirty entity-sets. We propose QuERy, a novel framework for integrating ER with query processing. The objective of QuERy is to *efficiently* and *accurately* answer complex Select-Project-Join (SPJ) queries issued on top of dirty data. The predicates in those queries may be associated with any attribute in the entity-sets being queried.

In particular, QuERy leverages the selectivities offered by the query predicates to reduce the amount of cleaning (by

BB, iPhone 6, 4.7" retina..., 415, 4.6, Apple, apple.com, USA
BB, iPhone 5, 4" retina..., 220, 3.9, Apple, apple.com, US
BB, Galaxy S5, 5.1" super..., 275, 4.3, Samsung, samsung.com, S. Korea
WM, Apple Inc., www.apple.com, USA, iPhone-VI, 550, 4.8
WM, Samsung, www.samsung.com, South Korea, Galaxy S-V, 180, 4.5
WM, Samsung, www.samsung.com, S. Korea, Galaxy S-III, 95, 3.7

Figure 1: A Collection of Raw Records

only deduplicating those parts of data that influence the query's answer) and thus, minimizes the total execution time of the query. We propose two variants of QuERy: lazy-QuERy and adaptive-QuERy. The former uses a lazy architecture that attempts to avoid cleaning until it is necessary for the system to proceed. The latter is an adaptive cost-based technique that tries to devise a good plan to decide when to perform cleaning. Both solutions rely on novel *polymorphic operators*, which are analogous to the common relational algebra operators (i.e., selections and joins) with one exception: they know how to test the query predicates on the dirty data prior to cleaning it. Specifically, these operators utilize *sketches* of data to perform inexact tests to decide whether parts of dirty data satisfy query predicates.

Overall, the **main contributions** of this paper are:

- We propose QuERy, a novel framework that integrates ER with query processing to answer complex SQL-style queries issued on top of dirty data (Sections 2 and 4).
- We introduce and formalize the notion of *polymorphic operators* – a key concept in QuERy (Section 5).
- We develop two different solutions: lazy-QuERy and adaptive-QuERy, which reap the benefits of evaluating the query predicates to minimize the query execution time (Sections 6 and 7).
- We conduct extensive experiments to evaluate the effectiveness of both lazy-QuERy and adaptive-QuERy solutions on real and synthetic datasets (Section 8).

2. PROBLEM SETUP

This paper addresses the problem of jointly cleaning and querying (potentially dirty) entity-sets in the context of generic SPJ SQL queries. To better motivate our work, before formalizing the problem and describing our solution, we first discuss a concrete example context wherein such cleaning challenges arise.

Consider an analyst wishing to explore popular electronic items produced in the "USA" that customers buy and write reviews about. The analyst identifies multiple relevant online data sources like Google Shopping, eBay, Walmart, Best Buy, Yelp, etc., which contain *raw* records describing (similar) entities. Suppose that the analyst chooses two such data sources: *Best Buy* (denoted by *BB*) and *Walmart* (denoted by *WM)*. Figure 1 shows a toy example of six raw records collected from these two sources. In general, such data sources may contain listings of several thousand entities[1]. Suppose that the analyst wishes to quickly identify cellphones that are listed on both data sources with at least 300 reviews, and that have been manufactured in the "USA".

In order to execute her analysis task, the analyst needs to utilize semi-automated tools (e.g., Data Wrangler [18] – an interactive tool that allows users to restructure input datasets prior to analysis) to convert the raw data into a form that is more convenient for data consumption.

In Figures 2 and 3, we present a transformed dataset instance that results from the raw records in Figure 1 by collating entities of the same type into a single entity-set

[1]Entities from the same source in Figure 1 are not duplicates. Our approach, however, does not make any assumption regarding the input data. Thus, entities from one source may or may not be duplicates.

block	source	c_id	c_name	m_id	c_reviews	c_ratings
C_1	*BB*	c_1	iPhone 6	m_1	415	4.6
	BB	c_2	iPhone 5	m_2	220	3.9
	WM	c_4	iPhone-VI	m_4	550	4.8
C_2	*BB*	c_3	Galaxy S5	m_3	275	4.3
	WM	c_5	Galaxy S-V	m_5	180	4.5
	WM	c_6	Galaxy S-III	m_6	95	3.7

Figure 2: Cellphones Entity-set (C)

block	source	m_id	m_name	m_url	m_country
M_1	*BB*	m_1	Apple	apple.com	USA
	BB	m_2	Apple	apple.com	US
	WM	m_4	Apple Inc.	www.apple.com	USA
M_2	*BB*	m_3	Samsung	samsung.com	S. Korea
	WM	m_5	Samsung	www.samsung.com	South Korea
	WM	m_6	Samsung	www.samsung.com	S. Korea

Figure 3: Manufacturers Entity-set (M)

(see Section 3.1). In this transformed representation, cellphone entities are stored in *Cellphones* entity-set (denoted by C), and manufacturer entities are stored in *Manufacturers* entity-set (denoted by M)[2]. Note that there exists only one manufacturer per cellphone and therefore, we can augment each cellphone entity with its corresponding *m_id*, *c_reviews*, and *c_ratings* values.

In such a transformed representation, since entities of the same type are collated into a single entity-set, standard ER algorithms like Swoosh [6], Sorted Neighbor (SN) [16], etc. can be used to cluster entities that co-refer into a single canonical representation[3]. For instance, entities $\{c_1, c_4\}$ and $\{c_3, c_5\}$ which refer to the same real-world object will be resolved and replaced by a common representation. Likewise, entities $\{m_1, m_2, m_4\}$ and $\{m_3, m_5, m_6\}$ are duplicates and will be replaced by their canonical representations. After this step, the data is ready for consumption and the analyst can pose her analysis task in the form of SQL-like queries.

Note that in the above approach data is cleaned *fully* prior to executing the query. However, the bulk of this cleaning, as we will see later, might be completely unnecessary since only a small portion of the data might influence the results of the user's analysis.

Based on this intuition, we propose QuERy, a framework that integrates ER with query processing into a single joint execution. In QuERy, after the analyst has restructured her data, she can postulate her queries over these entity-sets prior to cleaning them. Instead of cleaning the data fully beforehand and then executing the query, the QuERy framework exploits the query semantics to reduce the cleaning overhead by deduplicating only those parts of data that influence the query's answer. QuERy is agnostic to the specific technique of ER and thus, any ER algorithm (e.g., [6, 16]) can be used to clean the necessary data parts.

3. ER PRELIMINARIES

In this section, we first present common ER notation, and then discuss the standard phases of ER.

3.1 Entity-sets

Let $\mathcal{D}$ be a dataset instance that holds a number of entity-sets, $\mathcal{D} = \{R, S, T, \dots\}$. Each entity-set R contains a set of entities of the same type, $R = \{r_1, \dots, r_{|R|}\}$ where r_i represents the i^{th} entity in R and $|R|$ is its cardinality (s.t. $1 \leq i \leq |R|$). Entity-set R is considered *dirty* if at least

[2]The first column in Figures 2 and 3 refers to the "block-id" that results from partitioning the entities into blocks, see Section 3.2.

[3]Such ER algorithms typically assume that each entity-set contains a set of entities of the same type.

object	source	c_id	c_name	m_id	c_reviews	c_ratings
o_1^C	$\{BB, WM\}$	$\{c_1, c_4\}$	{iPhone 6, iPhone-VI}	$\{m_1, m_4\}$	415	4.8
o_2^C	BB	c_2	iPhone 5	m_2	220	3.9
o_3^C	$\{BB, WM\}$	$\{c_3, c_5\}$	{Galaxy S5, Galaxy S-V}	$\{m_3, m_5\}$	180	4.5
o_4^C	WM	c_6	Galaxy S-III	m_6	95	3.7

Figure 4: Object-set $\mathcal{O}^C$

object	source	m_id	m_name	m_url	m_country
o_1^M	$\{BB, WM\}$	$\{m_1, m_2, m_4\}$	Apple Inc.	www.apple.com	USA
o_2^M	$\{BB, WM\}$	$\{m_3, m_5, m_6\}$	Samsung	www.samsung.com	South Korea

Figure 5: Object-set $\mathcal{O}^M$

two entities $r_i, r_j \in R$ represent the same real-world object, and hence r_i and r_j are duplicates. The attributes in R are denoted as $A^R = \{R.a_1, \ldots, R.a_{|A^R|}\}$, where $|A^R|$ is the arity of R. Thus, the k^{th} entity in R is defined as $r_k = \{\nu_{k1}, \ldots, \nu_{k|A^R|}\}$, where ν_{ki} is the value of the i^{th} attribute in entity r_k (s.t. $1 \leq k \leq |R|$ and $1 \leq i \leq |A^R|$).

For instance, entity-set C, shown in Figure 2, is dirty since it contains two duplicate pairs $\{c_1, c_4\}$ and $\{c_3, c_5\}$. Likewise, entity-set M, presented in Figure 3, is dirty since manufacturers $\{m_1, m_2, m_4\}$ and $\{m_3, m_5, m_6\}$ are duplicates and refer to the same real-world objects.

3.2 Standard ER Phases

A typical ER cycle consists of two phases: a blocking phase and a deduplication phase.

3.2.1 Blocking Phase

Blocking is the main technique used to improve the efficiency of ER approaches. It divides entities of the same entity-set into (possibly overlapping) smaller blocks. Blocking methods use blocking functions, which are applied on one (or more) attribute(s) called *blocking key(s)*, to partition the entities such that (i) if two entities might co-refer, they will be placed together into at least one block and (ii) if two entities are not placed together into at least one block, they are highly unlikely to co-refer. In our approach, we divide each entity-set $R \in \mathcal{D}$ into a set of blocks $\mathcal{B}^R = \{R_1, \ldots, R_{|\mathcal{B}^R|}\}$ using one or more blocking functions.

For example, in Figure 2, we used a blocking function to partition the cellphones entity-set into blocks based on the first two letters of their names. The first column of the table in Figure 2 represents the "block-id" in which the entities reside. Entity-set C has two blocks C_1 and C_2.

3.2.2 Deduplication Phase

The goal of the (potentially expensive) deduplication phase is to detect, cluster, and then merge duplicate entities. It consists of three sub-phases: similarity computation, clustering, and merging, which can be intermixed.

Similarity Computation phase determines for each pair of entities within the same block whether they co-refer or not. This phase is often computationally expensive as it might require comparing every pair of entities in the same block using a compute-intensive application-specific *resolve* (match) function. In general, various ER algorithms use different techniques to perform this step. In our approach, any ER algorithm (e.g., Swoosh [6], Sorted Neighbor (SN) [16]) can be used to execute this step.

Clustering phase aims to accurately group duplicate entities in entity-set R into a set of non-overlapping clusters $\mathcal{C}^R = \{C_1^R, \ldots, C_{|\mathcal{C}^R|}^R\}$.

Merging phase combines entities of each individual cluster into a single object that will represent the cluster to the end-user or application in the final result. Let $\mathcal{O}^R = \{o_1^R, \ldots, o_{|\mathcal{O}^R|}^R\}$ be the object-set that results from merging clustering $\mathcal{C}^R = \{C_1^R, \ldots, C_{|\mathcal{C}^R|}^R\}$ such that each object $o_i^R \in \mathcal{O}^R$ represents the merging of cluster $C_i^R \in \mathcal{C}^R$.

A *merge* function $\text{M}^R(C_m^R)$ will consolidate the elements in cluster C_m^R to produce a new object o_m^R. To merge two duplicate entities $r_i, r_j \in C_m^R$, a *combine* function is used for each attribute in A^R. A combine function $\oplus_{R.a_\ell}$ takes two values of attribute $R.a_\ell \in A^R$ and outputs a single value. Such a combine function performs different operations based on the type of attribute $R.a_\ell$.

Note that the selection of these combine functions is domain-dependent and hence, should be done wisely by the analyst/user. If $R.a_\ell$ is a numeric attribute, $\oplus_{R.a_\ell}$ could be:

- ADD semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} + \nu_{j\ell}$,
- AVG semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \text{avg}(\nu_{i\ell}, \nu_{j\ell})$,
- MAX semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \max(\nu_{i\ell}, \nu_{j\ell})$,
- MIN semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \min(\nu_{i\ell}, \nu_{j\ell})$.

In general, the ADD and AVG semantics could be used when the objects are obtained from the same data source, yet their entities are split in parts. The MAX and MIN semantics could be used when objects are retrieved from different data sources, e.g., *c_reviews* and *c_ratings* in entity-set C.

If $R.a_\ell$ is a categorical attribute, $\oplus_{R.a_\ell}$ could be:

- UNION semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} \cup \nu_{j\ell}$,
- EXEMPLAR semantics: $\nu_{i\ell} \oplus \nu_{j\ell}$ chooses either $\nu_{i\ell}$ or $\nu_{j\ell}$ according to some policy.

The UNION semantics are utilized when the system needs to retain all possible values of some attribute, e.g., *c_name* in entity-set C. In contrast, the EXEMPLAR semantics are used when one value holds richer information than the other one, e.g., in the *m_name* attribute, the value "Apple Inc." dominates "Apple".

If $R.a_\ell$ is an identifier (e.g., *c_id*) or a reference attribute (e.g., *m_id* in entity-set C), $\oplus_{R.a_\ell}$ could be:

- UNION semantics: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{i\ell} \cup \nu_{j\ell}$.

Note that the aforementioned combine functions have the commutativity and associativity properties defined as:

1. Commutativity: $\nu_{i\ell} \oplus \nu_{j\ell} = \nu_{j\ell} \oplus \nu_{i\ell}$,
2. Associativity: $(\nu_{i\ell} \oplus \nu_{j\ell}) \oplus \nu_{k\ell} = \nu_{i\ell} \oplus (\nu_{j\ell} \oplus \nu_{k\ell})$.

Since these properties hold, the representation of the merged object (e.g., object o_m^R) will always be the same regardless of the merge order within C_m^R.

Figures 4 and 5 show the object-sets $\mathcal{O}^C = \{o_1^C, o_2^C, o_3^C, o_4^C\}$ and $\mathcal{O}^M = \{o_1^M, o_2^M\}$, which resulted from deduplicating entity-sets C and M using a traditional cleaning algorithm, e.g., [6, 16]. In Figures 4 and 5, we assume that the analyst picked (i) the MIN semantics to combine attribute *c_reviews*, (ii) the MAX semantics to combine attribute *c_ratings*, (iii) the EXEMPLAR semantics to combine attributes: *m_name*, *m_url*, and *m_country*, and (iv) the UNION semantics otherwise.

4. ER AND QUERY PROCESSING

In this section, we first introduce the concepts of queries and query trees in Section 4.1. Section 4.2 shows how to evaluate predicates applied to multi-valued attributes. In Section 4.3, we present a standard solution to answer queries on top of dirty data. Finally, we formally define our problem in Section 4.4.

4.1 Queries and Query Trees

We will consider *flat* SQL queries with AND as the only boolean connective in their qualification, similar to the following syntax: (SELECT *target-list* FROM $\mathcal{D}$ WHERE Φ),

```
SELECT  *
FROM    C AS C_x, C AS C_y, M
WHERE   C_x.m_id = M.m_id                                   // φ0
AND     C_x.c_name = C_y.c_name                             // φ1
AND     M.m_country = "USA"                                 // φ2
AND     C_x.c_source = "BB" AND C_x.c_reviews ≥ 300         // φ3
AND     C_y.c_source = "WM" AND C_y.c_reviews ≥ 300         // φ4
```

Figure 6: Query 1

where Φ denotes (i) the *equi-join*[4] predicate(s) connecting entity-sets in $\mathcal{D}$ and (ii) the optional selection predicates[5] applied to attributes in entity-sets in $\mathcal{D}$.

A flat SQL query is often represented by a SPJ query tree. The leaves of such a tree are relations while the non-leaf nodes are relational algebra operators, e.g., selections (σ), projections (π), joins ($\bowtie$), renames (ρ), etc. Each intermediate node encapsulates a single task that is required to execute the query. The edges of such a query tree represent data flow from the bottom to the top.

4.2 Evaluating Set Values

As presented in Section 3, if the UNION semantics are used to merge a non numeric attribute (viz., a categorical, an identifier, or a reference attribute), then the value of this attribute will be multi-valued in the form of a set, e.g., attributes *source*, *c_id*, *c_name*, and *m_id* in Figure 4. To evaluate predicate $\varphi_k \in \Phi$ defined on one of these attributes, we must overload its (=) operator as discussed next.

Let op_i and op_j be the two operands in predicate $\varphi_k : op_i = op_j$. The following four different cases exist since φ_k has two operands:

1. If both operands op_i and op_j are single-valued, then φ_k is `true` if $op_i = op_j$.
2. If operand op_i is a single value but operand op_j is a set, then φ_k is `true` if $op_i \in op_j$.
3. If operand op_i is a set but operand op_j is a single value, then φ_k is `true` if $op_j \in op_i$.
4. If both operands op_i and op_j are sets, then φ_k is `true` if $op_i \cap op_j \neq \{\}$.

4.3 Standard Solution

Let us use an illustrative example to present the standard solution of answering queries on top of dirty data. Suppose that a user interested in comparative shopping wishes to find popular cellphones that have been manufactured in the "USA" and are listed on two distinct data sources: *Best Buy* and *Walmart* with at least 300 reviews at each source. Query 1, shown in Figure 6, represents the user's interest.

The execution plan for Query 1, which is selected by some query optimizer, is shown in Figure 7. This plan is assumed to contain the best operators placement and join ordering. The result of executing this plan on the dirty entity-sets shown in Figures 2 and 3 prior to cleaning them is the empty set {}. However, this is incorrect since the first clean object {iPhone 6, iPhone-VI} is listed on both sources, it has more than 300 reviews at each source, and it has been manufactured in the "USA". Thus, it should be returned as the answer to Query 1.

The standard way to answer Query 1 is to first apply the blocking phase on the dirty entity-sets C and M, then to fully deduplicate C and M to create object-sets $\mathcal{O}^C$ and $\mathcal{O}^M$ (i.e., to obtain the tables in Figures 4 and 5), and finally, to compute the query over these object-sets. This corresponds to inserting the cleaning operator (denoted by δ) directly above the tree leaves (viz., entity-sets) as shown in Figure 8.

However, such an approach could be very expensive as it might clean unnecessary blocks. For instance, we note that no cellphones from C_2 could satisfy Query 1. This is because if we enumerate all potential objects inside C_2 (viz., $\{c_3\}$, $\{c_5\}$, $\{c_6\}$, $\{c_3, c_5\}$, $\{c_3, c_6\}$, $\{c_5, c_6\}$, and $\{c_3, c_5, c_6\}$) none of them will satisfy Query 1 as the maximum number of reviews of all potential objects inside C_2 is 275, which does not satisfy the reviews criteria, i.e., ≥ 300. Thus, all cellphones in C_2 will not be present in the query answer. As a result, deduplicating C_2 can be eliminated. By using a similar kind of reasoning, we can note that deduplicating M_2 is also unnecessary. Based on this intuition, we will build a principled solution for integrating ER with query processing in Sections 6 and 7 which demonstrates outstanding results, as shown in Section 8.

4.4 Problem Definition

Given a query Q, let $\mathcal{O}^f$ denote the set of objects that satisfy Q when all entity-sets in $\mathcal{D}$ are cleaned first. Also, let $\mathcal{O}^q$ be the set of objects returned by QuERy as the answer to Q. Then, we can formally define our problem as an optimization problem as follows:

Minimize: Execution time of Q
Subject to:
1. $\forall o \in \mathcal{O}^q$, o satisfies Q; // *Query satisfaction*
2. $\mathcal{O}^q \equiv \mathcal{O}^f$; // *Answer correctness*

5. POLYMORPHIC QUERY TREES

In this section we formalize the notion of *polymorphic operators*, which is a key concept in QuERy.

5.1 Polymorphic Operators

In order to correctly answer a query while performing only a minimal amount of cleaning, we introduce the concept of *polymorphic operators*. Such polymorphic operators are analogous to the common relational algebra operators (i.e., selections and joins) with one exception: they know how to test the query predicates on the dirty data prior to cleaning it. These operators are called polymorphic since they accept as input not only clean data (objects) as regular operators, but also dirty data (blocks).

Let us define these novel operators formally. Let R and S be two entity-sets in $\mathcal{D}$ that are split into two sets of blocks $\mathcal{B}^R = \{R_1, \ldots, R_{|\mathcal{B}^R|}\}$ and $\mathcal{B}^S = \{S_1, \ldots, S_{|\mathcal{B}^S|}\}$. Also, let $\mathcal{O}^R = \{o_1^R, \ldots, o_{|\mathcal{O}^R|}^R\}$ and $\mathcal{O}^S = \{o_1^S, \ldots, o_{|\mathcal{O}^S|}^S\}$ be the object-sets that would result if R and S are fully cleaned. Note that the objective of QuERy is to compute the query answer without fully computing $\mathcal{O}^R$ and $\mathcal{O}^S$, unless obviously if the query answer requires computing them fully.

To make our next definitions clear, let us denote the power set of block R_i as $\mathcal{P}(R_i)$. Power set $\mathcal{P}(R_i)$ is the set of all subsets of R_i, including the empty set and R_i itself. The merging of all entities in subset $R_i' \in \mathcal{P}(R_i)$ would form a *potential* object $\theta^{R_i'}$. Let the set of all potential objects inside block R_i be $\Theta^{R_i} = \{\theta_1^{R_i}, \ldots, \theta_{|\mathcal{P}(R_i)|}^{R_i}\}$. Note that not all potential objects in Θ^{R_i} will become real-world objects as a single entity cannot belong to two real-world objects.

A *polymorphic selection* is a unary operation, denoted by ς_{φ_k}, where φ_k is the propositional predicate. If the input to this operator is clean data (viz., object-set, say $\mathcal{O}^R$), then it acts as the normal selection operator σ_{φ_k} but applied to

[4] An equi-join is a special type of join that only uses the (=) operator in the join-predicate.

[5] If the selection predicate is applied to a non numeric attribute, then we solely consider the (=) operator.

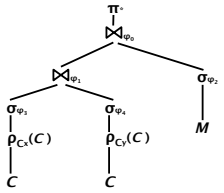

Figure 7: Query 1 Plan

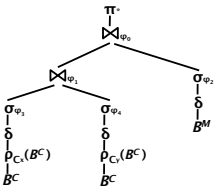

Figure 8: Clean First Plan

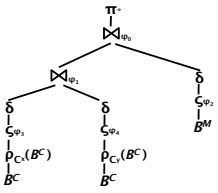

Figure 9: Plan A

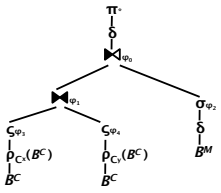

Figure 10: Plan B

objects instead of tuples. That is, it selects all those objects in $\mathcal{O}^R$ that satisfy φ_k. However, if its input is dirty data (viz., a set of blocks, say $\mathcal{B}^R$), then it selects all those blocks in $\mathcal{B}^R$ for which φ_k holds as defined next:

Definition 1. Block $R_i \in \mathcal{B}^R$ satisfies ς_{φ_k} if there exists a potential object $\theta^{R'_i} \in \Theta^{R_i}$ such that $\theta^{R'_i}$ satisfies σ_{φ_k}.

Intuitively, ς_{φ_k} would select those blocks $R_i \in \mathcal{B}^R$ that contain at least one potential object that satisfies σ_{φ_k}. For example, consider $\mathcal{B}^C = \{C_1, C_2\}$ from Figure 2, predicate $\varphi_i : C.c\_ratings = 4.8$, and combine function $\oplus_{c\_ratings} = \texttt{MAX}$. Block C_1 satisfies ς_{φ_i} since there exists a potential object $\theta^{C'_1} \in \Theta^{C_1}$ that satisfies σ_{φ_i}. Note that $\theta^{C'_1}$ resulted from merging subset $C'_1 = \{c_1, c_3\}$, where $C'_1 \in \mathcal{P}(C_1)$. The ratings value of $\theta^{C'_1}$ is 4.8 (i.e., $\max(4.6, 4.8) = 4.8$). In contrast, block C_2 does not satisfy ς_{φ_i} since no potential object $\theta^{C'_2} \in \Theta^{C_2}$ exists, which satisfies σ_{φ_i}. Thus, $\varsigma_{\varphi_i}(\mathcal{B}^C) = \{C_1\}$.

A *polymorphic join* is a binary operation, denoted by $\bowtie\!\!\!\!\blacktriangleright_{\varphi_k}$, where φ_k is the propositional predicate. Since $\bowtie\!\!\!\!\blacktriangleright_{\varphi_k}$ has two inputs, four cases exist. Case one is when both inputs to this operator are sets of clean objects, say $\mathcal{O}^R$ and $\mathcal{O}^S$, then it acts as the regular join operator $\bowtie_{\varphi_k}$ but applied to objects instead of tuples. That is, it joins all object pairs in $\mathcal{O}^R$ and $\mathcal{O}^S$ that satisfy φ_k. Case two occurs when the *left* input is dirty data (i.e., a set of blocks, say $\mathcal{B}^R$). The *left polymorphic join* operator, denoted by $\bowtie\!\!\!\!\blacktriangleright_{\varphi_k}$, joins all those block-object pairs for which φ_k holds as defined next:

Definition 2. Block $R_i \in \mathcal{B}^R$ and object $o_j^S \in \mathcal{O}^S$ satisfy $\bowtie\!\!\!\!\blacktriangleright_{\varphi_k}$ if there exists a potential object $\theta^{R'_i} \in \Theta^{R_i}$ that joins with o_j^S to form an object $\theta^{R'_i} o_j^S$ that satisfies $\bowtie_{\varphi_k}$.

In other words, $\bowtie\!\!\!\!\blacktriangleright_{\varphi_k}$ would associate those blocks $R_i \in \mathcal{B}^R$ that contain at least one potential object $\theta^{R'_i}$ with object o_j^S such that $\theta^{R'_i}$ joins o_j^S to satisfy $\bowtie_{\varphi_k}$.

Similarly, case three happens if the *right* input is a set of dirty blocks (say $\mathcal{B}^S$). The *right polymorphic join* operator, denoted by $\blacktriangleleft\!\!\!\bowtie_{\varphi_k}$, joins all those object-block pairs for which φ_k holds as defined next:

Definition 3. Object $o_j^R \in \mathcal{O}^R$ and block $S_i \in \mathcal{B}^S$ satisfy $\blacktriangleleft\!\!\!\bowtie_{\varphi_k}$ if there exists a potential object $\theta^{S'_i} \in \Theta^{S_i}$ that joins with o_j^R to form an object $o_j^R \theta^{S'_i}$ that satisfies $\bowtie_{\varphi_k}$.

Case four takes place when both inputs are sets of dirty blocks, then the polymorphic join operator $\blacktriangleleft\!\!\!\bowtie_{\varphi_k}$ joins all those block-block pairs for which φ_k holds as defined next:

Definition 4. Blocks $R_i \in \mathcal{B}^R$ and $S_j \in \mathcal{B}^S$ satisfy $\blacktriangleleft\!\!\!\bowtie_{\varphi_k}$ if there exists a potential object $\theta^{R'_i} \in \Theta^{R_i}$, a potential object $\theta^{S'_j} \in \Theta^{S_j}$, and the joined object $\theta^{R'_i}\theta^{S'_j}$ satisfies $\bowtie_{\varphi_k}$.

Intuitively, $\blacktriangleleft\!\!\!\bowtie_{\varphi_k}$ would associate those blocks $R_i \in \mathcal{B}^R$ that contain at least one potential object $\theta^{R'_i}$ with those blocks $S_i \in \mathcal{B}^S$ that contain at least one potential object $\theta^{S'_j}$ such that $\theta^{R'_i}$ joins $\theta^{S'_j}$ to satisfy $\bowtie_{\varphi_k}$. For instance, consider $\mathcal{B}^C = \{C_1, C_2\}$ and $\mathcal{B}^M = \{M_1, M_2\}$ from Figures 2 and 3, predicate $\varphi_j : C.m\_id = M.m\_id$, and combine function $\oplus_{m\_id} = \texttt{UNION}$. Blocks C_1 and M_1 satisfy $\bowtie\!\!\!\!\blacktriangleright_{\varphi_j}$ since potential object $\theta^{C'_1}$ that results from merging subset $C'_1 = \{c_1, c_2, c_4\}$ joins with potential object $\theta^{M'_1}$ that results from merging subset $M'_1 = \{m_1, m_2, m_4\}$. Note that $C'_1 \in \mathcal{P}(C_1)$ and $M'_1 \in \mathcal{P}(M_1)$. Also, note that the value of the *m_id* attribute in $\theta^{C'_1}$ is $\{m_1, m_2, m_4\}$ and the value of the *m_id* attribute in $\theta^{M'_1}$ is $\{m_1, m_2, m_4\}$.

Finally, the cleaning operator δ deduplicates the entities in the blocks using an ER algorithm, e.g., [6, 10].

5.2 Equivalent Polymorphic Query Trees

As discussed in Section 4.3, executing the plan shown in Figure 7 will return incorrect results because of the dirtiness in the data. To overcome this problem, a traditional solution will clean data first, which is semantically equivalent to inserting the deduplication operator δ directly above the tree leaves, see Figure 8. However, such an approach could be expensive as it might perform unnecessary cleaning and hence increase query execution time.

To return a correct result while trying to avoid unnecessary cleaning, we will augment the plan presented in Figure 7 with appropriate polymorphic operators. Clearly, many ways may exist in which we can add the polymorphic operators to such a plan. For instance, one plan may decide to perform polymorphic selections only and then clean all dirty blocks that pass such polymorphic selections, see Figure 9. The intuition for such a plan is that the polymorphic selections will be able to filter away some blocks without cleaning them. However, this strategy misses the opportunity to further prune unnecessary blocks which would have been pruned had we also considered the join predicate(s).

Another plan may choose to clean some entity-sets immediately while deferring the cleaning of other entity-sets to a later time. Figure 10 shows an example of such a plan where blocks from entity-set M are cleaned eagerly while the cleaning of blocks from entity-set C is deferred to a later time.

A different *lazy* plan may delay the cleansing of all dirty entity-sets to the end by inserting the cleaning operator δ above all selections and joins. Note that, in such a scenario, all selection and join operators that are originally applied on the dirty entity-set must be replaced with the corresponding polymorphic selections and joins, see Figure 11.

In general, we formally define the equivalence of polymorphic query trees as:

Definition 5. Two polymorphic query plans are said to be *equivalent* if the two plans return the *same* set of objects as their answer on every dataset instance $\mathcal{D}$.

In the subsequent sections, we will explain how to use the polymorphic operators to intermix query evaluation with ER to improve cleaning efficiency. In particular, we develop two different solutions: lazy-`QuERy` (Section 6) and adaptive-`QuERy` (Section 7) which utilize the query semantics to reap the benefits of early predicate evaluation while still minimizing unnecessary computation in the form of data cleaning.

6. LAZY-QUERY SOLUTION

In this solution, we develop a lazy architecture that attempts to delay cleansing of dirty entities as much as possible. The main idea in this approach is to try to avoid cleaning until it is necessary for the system to proceed. This architecture relies on the concepts of *polymorphic operators* and *sketches* to avoid unnecessary cleaning.

Conceptually, the lazy-QuERy solution can be viewed as consisting of the following steps:

1. *Create Blocks.* The approach starts, as most traditional ER approaches would; by partitioning entities of the same entity-set into (possibly overlapping) smaller blocks.
2. *Create Sketches.* In this step, the approach creates a sketch for each block to summarize its content. Sketches allow the polymorphic operators to perform an inexact test to decide whether a block satisfies a predicate or not without cleaning the block. Note that the sketches of each entity-set will be maintained in a separate LIFO stack. We denote the LIFO stack for entity-set E_i as $\mathcal{L}^{E_i}$.
3. *Query Plan Execution.* The approach evaluates the query tree using polymorphic operators' implementations based on the sketches. A block whose sketch does not satisfy the predicate will be discarded. A block whose sketch reaches the topmost operator (viz., passes all predicates) will be cleaned using a cleaning algorithm. The output of cleaning (i.e., clean objects) will be pushed back into the query tree to be evaluated. The algorithm terminates when there are no more sketches/objects to be tested.

6.1 Creating Sketches

Recall that a block R_i satisfies a polymorphic operator if there exists a potential object $\theta^{R'_i}$, which resulted from merging entities in subset $R'_i \in \mathcal{P}(R_i)$, that satisfies the common relational algebra operator. However, constructing the *power set* of block R_i in order to enumerate all potential objects inside R_i is exponential[6] and hence, impractical.

Consequently, the challenge translates into developing a good technique that allows polymorphic operators to perform their tests faster than the naive approach (viz., power set construction). The idea hinges on representing the values of the potential objects inside a block efficiently without constructing them.

A *sketch* is a concise representation of block contents. The key intuition behind sketches is to provide the polymorphic operators with the ability to quickly check if a block satisfies a predicate or not without cleaning the block. This test is considered a *safe* approximation: while it may return false positives, it never returns false negatives. That is, the polymorphic operator returns `false` only when all potential objects inside block R_i are guaranteed not to be a part of the query answer. In contrast, it returns `true` when at least one potential object inside R_i *might* be part of the answer. In our solution, we create a sketch $\mathcal{K}_i^R$ for each block R_i.

Before we formally define sketch $\mathcal{K}_i^R$, we need to introduce some auxiliary notation. Given the set of predicates Φ in query Q, let $P^{R_i} = \{R_i.a_1 \ldots, R_i.a_m\}$ be the set of attributes in R_i that are used in Φ, where $P^{R_i} \subseteq A^{R_i}$. Also, let $\mathcal{V}_j^{R_i} = \{\nu_{1j}, \ldots, \nu_{kj}\}$ represent the set of all values in block R_i for attribute $R_i.a_j$.

Sketch $\mathcal{K}_i^R$ is defined as $\mathcal{K}_i^R = \{K_1^{R_i}, \ldots, K_m^{R_i}\}$, where $K_j^{R_i}$ is a signature for attribute $R_i.a_j$. The value of signature $K_j^{R_i}$ represents the compact representation of the values of the $R_i.a_j$ attribute. The type of $K_j^{R_i}$ depends on the type of attribute $R_i.a_j$. Our approach computes signatures as follows. If $R_i.a_j$ is a numeric attribute, then $K_j^{R_i}$ is a *range* $[x, y]$. The values of x and y are computed based on the combine function $\oplus_{R_i.a_j}$ used to merge the values in $R_i.a_j$. That is, if $\oplus_{R_i.a_j}$ is:

1. `ADD`, then $x = \min(\mathcal{V}_j^{R_i})$ and $y = \sum_{\ell=1}^{k} \nu_{\ell j}$.
2. `MAX`, `MIN`, or `AVG`, then $x = \min(\mathcal{V}_j^{R_i})$ and $y = \max(\mathcal{V}_j^{R_i})$.

If $R_i.a_j$ is an identifier, a reference, or a categorical attribute, then the type of $K_j^{R_i}$ is a *set* U. Set U is computed as follows: $U = \bigcup_{\ell=1}^{k} \texttt{hash}(\nu_{\ell j})$, where `hash()` is a function that computes the hash value of $\nu_{\ell j}$.

The following example describes the concept of sketches. Consider entity-set C, Query 1, and combine functions: $\oplus_{source}$ = `UNION`, $\oplus_{c\_name}$ = `UNION`, $\oplus_{m\_id}$ = `UNION`, and $\oplus_{c\_reviews}$ = `MIN`. We create sketch $\mathcal{K}_1^C$ for block C_1 in Figure 2 as follows:

source	c_name	m_id	c_reviews
{hash(BB), hash(WM)}	{hash(iPhone 6), hash(iPhone 5), hash(iPhone-VI)}	{hash(m_1), hash(m_2), hash(m_4)}	[220, 550]

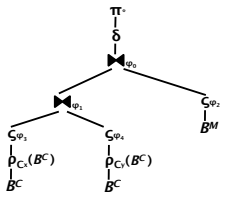

Figure 11: Lazy Plan

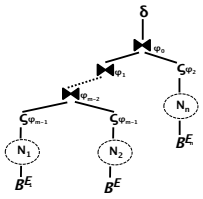

Figure 12: Decision Nodes

6.2 Query Plan Execution

The lazy-QuERy solution exploits the concept of "Equivalent Query Trees" to evaluate the query plan using polymorphic operators' implementations based on the blocks' sketches. It assumes the query tree that contains the best operator placement and join ordering is given. For example, the input to the lazy solution will be a plan similar to the "Query 1 Plan" in Figure 7. Given such a plan, the lazy solution substitutes each algebraic operator in the query tree, which is defined on a dirty entity-set, with its corresponding polymorphic operator. It then places the cleaning operator above all polymorphic selections and joins. As a result, a plan similar to the "Lazy Plan" in Figure 11 will be the plan that the lazy solution will create to execute the "Query 1 Plan" in Figure 7 on dirty data. The "Lazy Plan" is called *lazy* as it tries to delay the cleansing of dirty entities as much as possible. Its key insight is to try to avoid cleaning until it is necessary for the system to proceed.

QuERy employs the well-known database pipelining architecture. In pipelining, the output of one operator is passed to its parent without materializing the intermediate results. To support pipelining, the operators should implement the *Iterator Interface*, i.e., they implement three functions:

1. `open()`: initializes the iteration.
2. `getNext()`: calls operator-specific code to perform the operator's task. It also calls the `getNext()` function on each child operator to pull up the next item.
3. `close()`: ends the iteration when all output items have been produced through repeated calls to `getNext()`.

In QuERy, all polymorphic operators are implemented as *iterators*. An item (denoted by $\mathcal{I}$) in QuERy could be a sketch

[6] The complexity of constructing the power set of block R_i is $O(2^{|R_i|})$.

of a block, a clean object, or a composite item that results from joining two (or more) items. We refer to items that belong to a composite item as subitems. Note that, in QuERy, all query plans are either left (or right) deep plans since QuERy does not materialize the intermediate results.

The EXECUTE-PLAN(.) function, shown in Figure 13, begins when the topmost operator (viz., tree root[7]) calls open() to initialize the state of the iterator (Line 2). Then, it issues repeated calls to getNext() to pull up the next items (Lines 3–4). Each getNext() call performs two tasks: (i) it calls getNext() on each child operator and (ii) performs a specific task based on the operator type. If the operator is *select*, then it filters away items that do not satisfy the selection predicate. If the operator is *join*, then it combines items (from an entity-set) with other items (from another entity-set) that do satisfy the join predicate to form a composite item. Note that a block will be discarded if its sketch does not satisfy a predicate in the query tree.

Recall that, in this lazy solution, we place the cleaning operator above all polymorphic selections and joins and thus, an item will not reach the deduplicate operator unless it passes all predicates in the query tree. Such an item is, in fact, a composite item as it resulted from joining two (or more) items. If all subitems in this composite item are objects, then this composite item satisfies the query and hence, is added to the answer (Lines 5–6). However, if at least one subitem in this item is a sketch, then a *block-picking policy*[8] will choose a sketch from this item to clean its corresponding block (Lines 7–8). Note that this approach treats the cleaning strategy as a "black-box" and hence, any cleaning strategy (e.g. [6, 16], etc.) will suffice.

The output of cleaning (viz., clean objects) are pushed back into their appropriate stack (Line 9). It is important to note that the choice of a LIFO stack is critical since it provides higher priority to the clean objects compared to dirty blocks. Interestingly, this choice should intuitively reduce cleaning since the probability that an object will satisfy a predicate is usually much smaller when compared to the probability that a sketch will satisfy that exact predicate.

The algorithm terminates the iteration by calling close() when all items in the stacks are consumed (Line 11).

The idea of this function is shown next. Consider the plan in Figure 11. Its execution starts when the topmost operator (δ) calls getNext()[9] on its child. Subsequently, each operator calls getNext() (repeatedly) on its child operator(s). The bottommost operators call getNext() on *scan* operators to read the items from stacks $\mathcal{L}^{C_x}$, $\mathcal{L}^{C_y}$, and $\mathcal{L}^{M}$. Now, suppose that items $\mathcal{K}_1^{C_x}$, $\mathcal{K}_1^{C_y}$, and $\mathcal{K}_1^{M}$ pass all predicates in Figure 11 and thus, composite item $\mathcal{I} = \mathcal{K}_1^{C_x}\mathcal{K}_1^{C_y}\mathcal{K}_1^{M}$ reaches the deduplicate operator. Let the block-picking policy picks $\mathcal{K}_1^{C_x}$ to clean its block, i.e., C_1. This cleaning results in clean objects $o_1^{C_x}$ and $o_2^{C_x}$ which are pushed back to LIFO stack $\mathcal{L}^{C_x}$. Since QuERy uses LIFO stacks, objects $o_1^{C_x}$ or $o_2^{C_x}$ will be evaluated before $\mathcal{K}_2^{C_x}$. Finally, the algorithm terminates when $\mathcal{L}^{C_x}$, $\mathcal{L}^{C_y}$, and $\mathcal{L}^{M}$ are empty.

6.3 Correctness and Time Complexity

Correctness of Our Approach. From a theoretical perspective, QuERy framework guarantees that the following lemma holds trivially:

[7]In our discussion, we ignore projections and hence, we assume that the deduplicate operator is the tree root.

[8]Recall that, each sketch corresponds to a block and hence picking a sketch is, in fact, picking a block.

[9]Assume that open() is called implicitly before getNext().

```
EXECUTE-PLAN(root, L^{E_1}, ..., L^{E_n})
 1  Ans ← {}
 2  root.open()
 3  I ← root.getNext()
 4  while I ≠ null
 5     if ∀ι ∈ I, isObject(ι) then //all subitems in I are objects
 6        Ans ← Ans ∪ I
 7     else                          //at least one item is a sketch
 8        O^{E_j} ← PICK-AND-CLEAN-BLOCK(I)
 9        L^{E_j}.push(O^{E_j})
10     I ← root.getNext()
11  root.close()
12  return Ans
```

Figure 13: EXECUTE-PLAN(.) Function

Lemma 1. For any flat query Q, the object-set $\mathcal{O}^q$ returned by QuERy as the answer to Q is exactly equal to the object-set $\mathcal{O}^f$ returned by the standard solution (see Section 4.3) as the answer to Q.

Time Complexity of Our Approach. The polymorphic operators in QuERy work with any implementation of the corresponding common relational algebra operators and thus, does not influence the time complexity of such operators; except for the difference in the cost of evaluating the associated predicates. Note that costs of block-to-block, block-to-object and object-to-object tests could be different. In the last case, the cost is exactly equal to that of common implementations. The predicate cost, however, is (relatively) more in the case of block-to-(block/object) tests (though the number of tests is much smaller). For more details see [1].

6.4 Discussion

In this section, we briefly explain a few interesting points related to blocking.

Multiple blocking functions. ER techniques typically use several blocking functions to ensure that all the likely matching entities are compared, improving the quality of the result. Our solution deals with such a case by creating a separate set of LIFO stacks for each blocking function. Next, it considers each set of stacks independently.

Blocking key sketch optimization. An important (and frequent) special case takes place when there is a predicate defined on an attribute that was selected as a blocking key to partition the entity-set. In this case, the value of the signature for this attribute is chosen to be equal to the value returned by the blocking procedure (such a value is denoted by BKV). For instance, the second join predicates in Query 1 is $\varphi_1 : C_x.c\_name = C_y.c\_name$ and entity-set C in Figure 2 is divided based on the *c_name* attribute. Thus, the value of signature $K_{c\_name}^{C_1}$ is "iP". Note that this special case allows for even more efficient block-to-block join processing compared to using a union sketch, as we present in Section 8.

Block-picking policy. A block-picking policy selects one block from a composite item (that reached the deduplicate operator) to clean it. Intuitively, such a policy should choose a block that may reduce the query execution time. Many different policies can be used. We classify such policies into three main classes. The first class picks a block based on its location in the composite item. For example, a policy from this class may pick and clean the leftmost (or rightmost) available block first. The second class chooses a block to clean based on its size. For instance, it selects the smallest block available in the composite item. The third class picks a block to clean based on the selectivities of the predicates in the query (see Section 7). We have experimented with different policies from each class. The one that has demonstrated the best results is based on picking the rightmost available block from the composite item first. The block-picking policy is not our focus in this paper.

7. ADAPTIVE-QUERY SOLUTION

The previous solution is considered lazy since it tries to delay the cleaning of dirty entities as much as possible. While such an approach will reduce the cost of cleaning, it might increase the cost of processing the query. For example, assume that all sketches end up reaching the deduplicate operator (viz., the topmost operator in Figure 11), meaning that their corresponding blocks need to be cleaned. In this case, the time spent in trying to filter away these blocks is wasted. In fact, cleaning these blocks *eagerly* (without passing their sketches up the tree) might be more efficient.

To address this issue, we implement a different solution which is an *adaptive* cost-based approach that, given a query tree (with polymorphic operators) and dirty entity-sets, can devise a good plan to simultaneously clean and process the query. The key intuition hinges on placing *decision nodes* as the bottommost nodes in the query tree, as presented in Figure 12. The task of such decision nodes is to decide if eagerly cleaning some dirty blocks is more efficient (in terms of the overall query execution time) than delaying their cleansing until the last stage as in the lazy solution. The conjecture of placing these nodes at the bottom is to allow adaptive-`QuERy` to make the "cleaning a block eagerly versus passing it up the tree" decision, from the start of query execution time. Note our adaptive solution is general, and it does not require such nodes to be placed at the bottom.

Our adaptive cost-based solution consists of two steps. In the first step, we use a sampling technique to collect different statistics (e.g., selectivities of predicates, cost of join, etc.). In the second step, the decision nodes utilize these statistics to make their smart decisions.

7.1 Sampling Phase

The first step in our adaptive solution is to employ a sampling phase to collect various statistics such as: predicates' selectivities, cost of cleaning, cost of block-to-block join test, etc. The basic idea is to use two fractions f_1 and f_2, where $0 < f_1, f_2 < 1$, to control how many input blocks will be consumed (either cleaned or passed up the tree for evaluation) during this phase. In particular, the sampling phase begins by cleaning $f_1.|\mathcal{B}_{total}|$ blocks, where $|\mathcal{B}_{total}|$ is the total number of blocks in all entity-sets in dataset $\mathcal{D}$. For instance, consider dataset $\mathcal{D} = \{C, M\}$. Let $|\mathcal{B}^C| = |\mathcal{B}^M| = 1,000$ blocks and $f_1 = 10\%$. In this case, the sampling phase begins by eagerly cleaning $0.1 \times (1,000 + 1,000) = 200$ blocks. This early cleaning allows our adaptive-`QuERy` solution to estimate the following values:

1. The average cost of resolving two entities in entity-set R, denoted by $cost(resolve(R))$.
2. The average cost of joining a clean object with a clean object, denoted by $cost(OOJoin)$.
3. The clean object selectivity of each predicate φ_i in the query tree, denoted by $\mathbb{S}_{\mathcal{O}}(\varphi_i)$.

Subsequently, fraction f_2 is utilized (by our solution) to ensure that at least $f_2.|\mathcal{B}_{total}|$ blocks will be passed up the tree for evaluation. For example, if $f_2 = 20\%$, then $0.2 \times (1,000 + 1,000) = 400$ blocks will be sent up the tree for evaluation. This block-evaluation authorizes our solution to estimate the next set of values:

1. The average cost of joining a block sketch with a block sketch, denoted by $cost(BBJoin)$.
2. The average cost of joining a block sketch with an object, denoted by $cost(BOJoin)$.
3. The dirty block selectivity of each predicate φ_i in the query tree, denoted by $\mathbb{S}_{\mathcal{B}}(\varphi_i)$.

If predicate φ_i is associated with a *select* operator, then we calculate $\mathbb{S}_{\mathcal{B}}(\varphi_i)$ and $\mathbb{S}_{\mathcal{O}}(\varphi_i)$ as follows: $\mathbb{S}_{\mathcal{B}}(\varphi_i) = \frac{|\mathcal{B}_s(\varphi_i)|}{|\mathcal{B}_t(\varphi_i)|}$ and $\mathbb{S}_{\mathcal{O}}(\varphi_i) = \frac{|\mathcal{O}_s(\varphi_i)|}{|\mathcal{O}_t(\varphi_i)|}$, where $|\mathcal{B}_s(\varphi_i)|$ ($|\mathcal{O}_s(\varphi_i)|$) is the number of blocks (objects) that satisfy predicate φ_i and $|\mathcal{B}_t(\varphi_i)|$ ($|\mathcal{O}_t(\varphi_i)|$) is the number of blocks (objects) that was tested by predicate φ_i. For instance, if 50 blocks (out of 200 blocks tested) satisfy ς_{φ_3}, then, $\mathbb{S}_{\mathcal{B}}(\varphi_3) = \frac{50}{200} = 0.25$.

Yet, if predicate φ_i is associated with a *join* operator, then we estimate $\mathbb{S}_{\mathcal{B}}(\varphi_i)$ and $\mathbb{S}_{\mathcal{O}}(\varphi_i)$ as follows: $\mathbb{S}_{\mathcal{B}}(\varphi_i) = \frac{|\mathcal{B}_s(\varphi_i)|}{|\mathcal{B}_{t_1}(\varphi_i)|.|\mathcal{B}_{t_2}(\varphi_i)|}$ and $\mathbb{S}_{\mathcal{O}}(\varphi_i) = \frac{|\mathcal{O}_s(\varphi_i)|}{|\mathcal{O}_{t_1}(\varphi_i)|.|\mathcal{O}_{t_2}(\varphi_i)|}$, where $|\mathcal{B}_{t_1}(\varphi_i)|$ ($|\mathcal{O}_{t_1}(\varphi_i)|$) is the size of the first input set of blocks (clean objects) and $|\mathcal{B}_{t_2(\varphi_i)}|$ ($|\mathcal{O}_{t_2}(\varphi_i)|$) is the size of the second input set of blocks (clean objects) of the join operator.

During this sampling phase, we also calculate an important statistic: the probability of a given block R_i passing all predicates in the query tree and hence, reaching the deduplicate operator[10]. This probability, denoted by $\mathbb{P}(R_i)$, is computed as follows: $\mathbb{P}(R_i) = \frac{|\mathcal{B}^R_{top}|}{|\mathcal{B}^R_{read}|}$, where $|\mathcal{B}^R_{top}|$ is the number of blocks from entity-set R that reached the top operator and $|\mathcal{B}^R_{read}|$ is the number of blocks from R that were read during the sampling phase.

Intuitively, to better estimate the previous values, we would like to read a comparable number of blocks from each entity-set in $\mathcal{D}$. For instance, a join algorithm such as *tuple-based nested-loop*, where all tuples of the inner entity-set are read before reading the second tuple from the outer entity-set, will not be appropriate. Therefore, we implement the *ripple join* algorithm [13] as our join algorithm. This join algorithm aims to draw tuples (i.e., items) from entity-sets at the same rate and hence, should allow for better estimations.

7.2 Adaptive Cost-based Cleaning

Based on the statistics computed in the sampling phase, the decision nodes in our adaptive-`QuERy` solution use the decision plane, presented in Figure 14, to compare the cost of cleaning block R_i eagerly ($CleanNow(R_i)$) versus the cost of evaluating R_i's sketch in the query tree ($Evaluate(\mathcal{K}^R_i)$).

The decision whether to $CleanNow(R_i)$ or $Evaluate(\mathcal{K}^R_i)$ relies on the following four values:

1. The probability of block R_i reaching the deduplicate operator, denoted by $\mathbb{P}(R_i)$.
2. The cost of cleaning block R_i, denoted by $clean(R_i)$[11].
3. The cost of evaluating block R_i's sketch ($\mathcal{K}^R_i$) in the query tree, denoted by $tree(\mathcal{K}^R_i)$. We compute $tree(\mathcal{K}^R_i) = tree(\mathcal{K}^R_i) = \mathcal{T}^B.cost(BBJoin) + \mathcal{T}^O.cost(BOJoin)$, where $\mathcal{T}^B$ ($\mathcal{T}^O$) is the estimated number of block-to-block (block-to-object) join tests that will be performed to check if sketch $\mathcal{K}^R_i$ will reach the top operator.
4. The cost of evaluating R_i's cleaned objects ($\mathcal{O}^{R_i}$) in the tree, denoted by $tree(\mathcal{O}^{R_i})$. We compute $tree(\mathcal{O}^{R_i}) = tree(\mathcal{O}^{R_i}) = |R_i|(\mathcal{T}^B.cost(BOJoin) + \mathcal{T}^O.cost(OOJoin))$, where $|R_i|$ is the size of block R_i.

For instance, if $\mathbb{P}(R_i)$ is high and both $clean(R_i)$ and $tree(\mathcal{O}^{R_i})$ are low, then it might be more tempting to clean R_i immediately instead of deferring its cleaning with little hope of it being discarded.

The MAKE-A-DECISION(.) function, shown in Figure 15, presents the method a decision node employs to make its choice using the decision plane. It starts by computing

[10] Recall that, similar to the lazy solution, this solution places the cleaning operator above all polymorphic selections and joins.

[11] The cost of $clean(R_i)$ is algorithm-dependent (see Section 8.1).

Decision	$\mathbb{P}(R_i)$	$1-\mathbb{P}(R_i)$
$CleanNow(R_i)$	$clean(R_i)+tree(\mathcal{O}^{R_i})$	$clean(R_i)+tree(\mathcal{O}^{R_i})$
$Evaluate(\mathcal{K}_i^R)$	$clean(R_i)+tree(\mathcal{O}^{R_i})+tree(\mathcal{K}_i^R)$	$tree(\mathcal{K}_i^R)$

Figure 14: Decision plane

the cost of $CleanNow(R_i)$, which corresponds to the cost of cleaning block R_i plus the cost of evaluating the clean objects which resulted from cleaning R_i in the query tree (Line 1). It also computes the cost of $Evaluate(\mathcal{K}_i^R)$, which corresponds to computing the cost of two cases (Line 2). On the one hand, if the sketch passes all predicates and thus reaches the deduplicate operator, then the cost of evaluating it in the query tree is equal to the summation of the following three costs (i) the cost of evaluating this sketch in the query tree, (ii) the cost of cleaning the block that corresponds to this block, and (iii) the cost of testing the clean objects that resulted from cleaning its block. On the other hand, if the sketch is eliminated by one predicate in the query tree, then its evaluation cost is only the cost of testing it in the query tree. Finally, the algorithm picks the decision with the least amount of cost (Lines 3–5).

The idea of this function is shown in the next example. Suppose that sketch $\mathcal{K}_i^C$ (that corresponds to block C_i) is returned by the *scan* operator. Before passing $\mathcal{K}_i^C$ up the tree, the MAKE-A-DECISION(.) function wants to check if cleaning it eagerly is more efficient. It uses some costs that were estimated in the sampling phase. Let $clean(C_i)$ = 4.95 ms, $tree(\mathcal{K}_i^C)$ = 0.024 ms, $tree(\mathcal{O}^{C_i})$ = 0.16 ms, and $\mathbb{P}(C_i)$ = 0.25. The MAKE-A-DECISION(.) function computes $CleanNow(C_i)$ = 4.95 + 0.16 = 5.11 ms and $Evaluate(\mathcal{K}_i^C)$ = 0.25 × (4.95 + 0.16 + 0.024) + 0.75 × 0.024 = 1.3 ms. In this case, since $CleanNow(C_i) > Evaluate(\mathcal{K}_i^C)$, the MAKE-A-DECISION(.) function will pass up $\mathcal{K}_i^C$ for evaluation.

8. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the efficiency of our proposed approaches on real and synthetic datasets.

8.1 Experimental Setup

Solutions. In our experiments, we compare the following three solutions: (i) the standard solution (`SS`), presented in Section 4.3, (ii) the lazy-`QuERy` solution (`LQS`), explained in Section 6, and (iii) the adaptive-`QuERy` solution (`AQS`), described in Section 7.

Cleaning Algorithm. To deduplicate a dirty block $R_k \in \mathcal{B}^R$, we resolve any two entities $r_i, r_j \in R_k$ to try to decide whether they are duplicates or not as in [19]. We next cluster the duplicate pairs using the well-known transitive closure algorithm [14].

The cost of cleaning block R_k is algorithm-dependent. Thus, for this algorithm, we compute this cost as: $clean(R_k) = cost(resolve(R)).\frac{|R_k|.(|R_k|-1)}{2}$, where $cost(resolve(R))$ is the average cost of resolving two entities in entity-set R and $|R_k|$ is the size of block R_k.

8.2 Performance Factors

We can split the end-to-end execution time of `SS`, `LQS`, and `AQS` into two main parts: (i) the time spent at cleaning the dirty blocks and (ii) the time spent at processing the input items (viz., blocks and object). As we discuss next, several factors are expected to affect the performance of the three solutions. In this section, we summarize the expected effects of each factor prior to testing and validating their effects experimentally in the following sections.

MAKE-A-DECISION(R_i)
1 $CleanNow(R_i) \leftarrow clean(R_i) + tree(\mathcal{O}^{R_i})$
2 $Evaluate(\mathcal{K}_i^R) \leftarrow \mathbb{P}(R_i) * (clean(R_i) + tree(\mathcal{O}^{R_i}) + tree(\mathcal{K}_i^R))$
$+(1-\mathbb{P}(R_i)) * tree(\mathcal{K}_i^R)$
3 **if** $CleanNow(R_i) < Evaluate(\mathcal{K}_i^R)$ **then**
4 $\mathcal{O}^{R_i} \leftarrow$ CLEAN-BLOCK(R_i)
5 **else** $Evaluate(\mathcal{K}_i^R)$

Figure 15: MAKE-A-DECISION() Function

Block selectivity ($\mathbb{S}_{\mathcal{B}}$)**.** `SS` is not affected by the block selectivity because it does not deal with blocks. In contrast, `LQS` is affected in the following fashion. When $\mathbb{S}_{\mathcal{B}}$ is *high*, a small number of blocks will satisfy the query and thus, the time spent at evaluating the query and at cleaning the dirty blocks will decrease. However, when it is *low*, a larger number of blocks will satisfy the query. In general, the processing of the dirty blocks will be mostly overhead as most blocks will not be discarded. Therefore, the time spent at both evaluating the query and at cleaning the dirty blocks will increase. In addition, it affects `AQS` in the following way. When $\mathbb{S}_{\mathcal{B}}$ is *high*, `AQS`'s performance will be similar to `LQS`. However, when it is *low*, the performance of `AQS` will usually be similar to `SS`.

Object selectivity ($\mathbb{S}_{\mathcal{O}}$)**.** Object selectivity affects all three solutions similarly. When it is *high*, a small number of objects will satisfy the query and thus, the time to process the query will decrease. However, when $\mathbb{S}_{\mathcal{O}}$ is *low*, a larger number of objects will satisfy the query, and thus the time to evaluate the query will increase.

Cleaning Cost ($cost(resolve)$)**.** All three solutions are affected by this factor. Clearly, when the cost of cleaning is *high*, the time spent at cleaning the dirty blocks will be more than the time spent at cleaning them when it is *low*. Note that `SS` will suffer the most (compared to `LQS` and `AQS`) because it will always clean all blocks.

BBJoin Cost ($cost(BBJoin)$) **/ BOJoin Cost** ($cost(BOJoin)$)**.** `SS` is not affected by these two costs since it does not deal with blocks. In contrast, `LQS` is affected in the following way. When the costs of BBJoin and BOJoin are *low*, the overhead of evaluating the blocks will be low, and vice versa. Moreover, they affect `AQS` in the following fashion. When $cost(BBJoin)$ and $cost(BOJoin)$ are *low*, `AQS`'s performance will often mimic `LQS`. However, when they are *low*, its performance will usually imitate `SS`.

OOJoin Cost ($cost(OOJoin)$)**.** This cost affects all three solutions similarly. When $cost(OOJoin)$ is *high*, the time spent in processing the query will increase. However, when it is *low*, the time spent in evaluating the query will decrease.

Overall, `LQS` is expected to perform excellently when $\mathbb{S}_{\mathcal{B}}$ is high and/or $cost(BBJoin)$ and $cost(BOJoin)$ are low. We expect `LQS` to perform reasonably (still better than `SS`) when the costs of BBJoin and BOJoin tests are lower than the cleaning cost (of blocks that do not satisfy the query) plus the cost of evaluating the objects inside them.

Moreover, `AQS` is expected to perform competently when `LQS` performs well. In addition, it is expected to overcome the difficulties that `LQS` may face when the costs of BBJoin and BOJoin tests are higher than the cleaning cost (of blocks that do not satisfy the query) plus $cost(OOJoin)$) for the the objects inside these blocks.

8.3 Products Dataset Experiments

In this section, we evaluate the efficacy of our approaches on a real electronic products dataset collected from two different data sources: *Best Buy* and *Walmart*. We obtained a subset of 89,070 raw records from these sources. To enrich our data further, we collected 1,237 raw records from

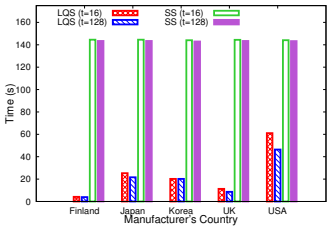

Figure 16: LQS vs. SS [Time]

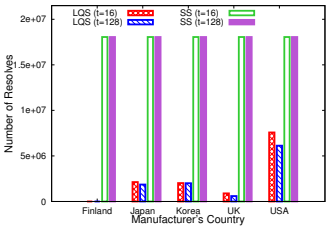

Figure 17: LQS vs. SS [#ℜ()]

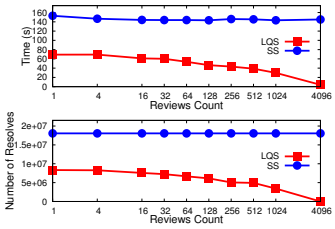

Figure 18: Selectivity Effects

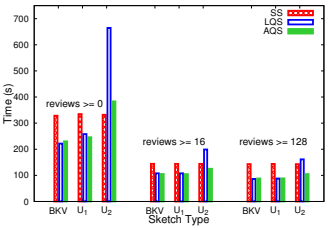

Figure 19: AQS Performance

Wikipedia regarding the product manufacturers.

Using basic data wrangling techniques, we restructured these raw records into dataset $\mathcal{D} = \{C, M\}$. Entity-set C contains entities that describe electronic products and entity-set M holds entities that describe their manufacturers. The dataset schema is similar to that of Figures 2 and 3. The cardinalities of entity-sets are $|C| = 89,070$ and $|M| = 2,144$.

Each entity-set is partitioned into a set of blocks. We use a blocking function to split entity-set C into a set of blocks (viz., $\mathcal{B}^C$) based on the first *six* letters of the products' names. In addition, we use a blocking function that partitions entity-set M into a set of blocks (i.e., $\mathcal{B}^M$) based on the first *two* characters of the manufacturers' names. The sizes of the resulted block-sets are $|\mathcal{B}^C| = 9,742$ and $|\mathcal{B}^M| = 292$.

To resolve a pair of entity-set C, we use the Edit-Distance algorithm to compute the similarity between the products' names. If this similarity is sufficient, then the resolve function declares the two products to be duplicates. In addition, to resolve entities from entity-set M, we utilize Jaro-Winkler distance to compare the names of the manufacturers.

To merge duplicate pairs from entity-sets C and M, we select the `MAX` semantics to combine all numeric attributes and the `UNION` semantics to combine all non numeric attributes.

Finally, we compute sketches for entity-set C as:

source	c_name	m_id	c_reviews
$\bigcup_{\ell=1}^{m}$ hash($\nu_{\ell j}$)	BKV	$\bigcup_{\ell=1}^{m}$ hash($\nu_{\ell j}$)	$[\min(\mathcal{V}_j^{C_i}), \max(\mathcal{V}_j^{C_i})]$

We also compute sketches for entity-set M as:

m_id	m_country
$\bigcup_{\ell=1}^{m}$ hash($\nu_{\ell j}$)	$\bigcup_{\ell=1}^{m}$ hash($\nu_{\ell j}$)

Experiment 1 (Lazy-QuERy Solution vs. Standard Solution). In this experiment, we use a set of queries similar to Query 1 to compare our lazy-QuERy solution (LQS) with the standard solution (SS) in terms of their end-to-end running time and the number of resolves called. Figure 16 plots the actual end-to-end execution time of both solutions for five different countries (viz., "Finland", "Japan", "Korea", "UK", and "USA") using two different *c_reviews* values (viz., $t = 16$ and $t = 128$). Figure 17 is similar to Figure 16 but plots the number of resolve calls instead of the execution time. Note that the histograms in the two figures are identical, thus demonstrating that calling resolve (and not query evaluation) is the bottleneck of both solutions in this test.

As expected, LQS is both faster and issues fewer resolves than SS. This is due to its awareness of the query which gives it the ability to discard blocks (whose sketches do not satisfy one of the query predicates), resulting in savings in resolves as such blocks do not require cleaning.

In Figure 16, when *m_country* = "Finland" and *c_reviews* ≥ 16 (or *c_reviews* ≥ 128), LQS takes only 4 seconds to evaluate the query and return the answer while SS takes more than 140 seconds to do so. This huge savings is due to the fact that, in our dataset, there is only one manufacturer that is located in "Finland" and hence a small number of blocks will require cleaning. However, there are more manufacturers that are located in "Japan", for instance, and thus LQS takes more than 20 seconds to answer such a query.

Experiment 2 (Selectivity Effects). Figure 18 utilizes a set of queries, similar to Query 1, to study the effects of the query selectivity on LQS as well as on SS. To control the query selectivity, we fix the manufacturer's country (i.e., *m_country* = "USA") and vary the *c_reviews* values (viz., 1, 4, ..., 4,096). Note that Figure 18 uses a log-lin scale plot and the selection of different *c_reviews* values is done carefully to show the entire spectrum of LQS's behavior.

As in Experiment 1, herein, the cost of calling resolve is also the dominant factor of the overall execution time. Hence, the end-to-end execution time of both solutions depends heavily on the number of blocks that will be cleaned.

The number of resolves invoked by LQS and obviously its query execution time (since resolves are the dominant factor) are relative to the number of blocks that satisfy the query. For instance, the number of blocks that satisfy predicate $\varphi_i : c\_reviews \geq 1$ is almost equal to the number of blocks that satisfy predicate $\varphi_j : c\_reviews \geq 4$; and hence, the LQS takes almost equal time to answer these two queries. In contrast, the block count that satisfies $\varphi_i : c\_reviews \geq 1$ is considerably more than the block count that satisfies $\varphi_k : c\_reviews \geq 128$ and hence, LQS solution takes considerably less time to answer the second query. In short, in case of LQS, whenever the number of blocks that satisfy the query (and hence require cleaning) increases, the total execution time to answer the query will increase, and vice versa.

Note that the query selectivity effects are not noticed on SS (in this experiment) since the cleaning cost overshadows the cost of query processing.

Experiment 3 (Execution Time Breakdown). LQS *almost* always outperforms SS as shown in Experiments 1 and 2. However, there are some cases in which SS performs better than LQS. Such cases usually occur when (i) the query selectivity is low (viz., most blocks will require cleaning) and/or (ii) *cost*(*BBJoin*) tests is high.

To present such a case, we use the following query: (`SELECT` $*$ `FROM` C `AS` C_x, C `AS` C_y `WHERE` $C_x.c\_name = C_y.c\_name$ `AND` $C_x.c\_source =$ "*BB*" `AND` $C_x.c\_reviews \geq t$ `AND` $C_y.c\_source =$ "*WM*" `AND` $C_y.c\_reviews \geq t$), where we choose the value of t to be equal to 0. Note that the selectivity in this query is only offered by the join predicate and hence its selectivity is low. In addition, we experiment with three different *BBJoin* tests of various costs. The first test (denoted by *BKV*) is the least expensive one. It uses the blocking key value (BKV) as the signature value for attribute *c_name*. It decides if two blocks join or not by comparing their BKVs. Test two (denoted by U_1) is more expensive than test one. The signature of attribute *c_name*, in this case, is equal to a set of hashed strings. It checks if two blocks join or not

	SS	LQS (BKV)	LQS (U_1)	LQS (U_2)
Total Time (s)	328.586	221.719	258.023	664.690
Initialization Time	1.148%	1.715%	1.495%	0.578%
Resolution Time	40.579%	49.988%	42.519%	16.440%
BBJoin Time	0%	2.374%	6.00%	63.043%
BOJoin Time	0%	16.128%	23.331%	9.482%
OOJoin Time	44.947%	19.222%	17.397%	6.791%
Iteration Time	13.299%	10.573%	9.258%	3.666%

Figure 20: Execution Time Breakdown

by efficiently computing the intersection between two sets of integers. The third test (denoted by U_2) is the most expensive one. In this case, the signature of attribute *c_name* is a set of strings. It checks if two blocks join or not by exhaustively comparing the string values in the two sets. In short, the cost of testing $U_2 >> U_1 > BKV$.

To conduct this experiment, we ran each of the four approaches shown in Figure 20 multiple times and recorded the average total execution time (the first row in Figure 20). We also computed the breakdown of that execution time which consists of: the times spent at (i) blocking and creating sketches if necessary, (ii) resolving dirty blocks, (iii) joining dirty blocks, (iv) joining blocks with objects, (v) joining clean objects, (vi) and running the iterator interface. Results showed that running LQS using the second block-to-block join test (viz., U_1) takes 258.023 seconds. In particular, we show that resolving the dirty blocks takes 42.519% of the total execution time while joining them takes only 6.00%, demonstrating that resolving blocks is more expensive than joining them in this case.

As shown in Figure 20, SS outperforms LQS when we use the third *BBJoin* test (viz., U_2). This is because *cost*(*BBJoin*) dominates the cost of cleaning and hence, it might be better to clean the blocks eagerly as we show next.

Experiment 4 (Adaptive-QuERy Solution). Figure 19 studies the performance of our adaptive-QuERy solution (AQS). We, herein, use a set of queries similar to the query presented in Experiment 3. In this experiment, we choose three different values (i.e, 0, 16, and 128) for variable t and we continue to experiment with the three different block-to-block join tests (viz., BKV, U_1, and U_2). In addition, we experimentally set the fraction values to: $f_1 = 5\%$ and $f_2 = 20\%$.

In this experiment we study three interesting cases. The first case occurs when the resolution cost dominates the block-to-block join test cost (i.e., BKV and U_1). In such a case, AQS acts similarly to LQS and hence, it always outperforms *SS* and has almost identical performance to LQS, see Figure 19. Case two takes place when the the block-to-block join test cost dominates the resolution cost as in U_2 and the query selectivity is not very low (viz., $t = 16$ or $t = 128$). Herein, AQS surpasses both LQS and SS. This is due to the fact that, AQS often makes the correct decision of whether to clean the block eagerly or pass it up for evaluation based on the statistics collected in the sampling phase. The third case happens when the the block-to-block join test cost dominates the resolution cost (i.e., U_2) and the query selectivity is low (viz., $t = 0$). In this case, AQS outperforms LQS since it has the ability to utilize the knowledge that cleaning the block eagerly is more efficient than passing it up the tree. However, it requires some time (i.e., the sampling phase time) to discover this fact, and hence SS outperforms AQS in this extreme case.

Note that, unlike Experiment 2, herein the impact of the query selectivity is noticed on SS since the cleaning cost does not overshadow the cost of query processing.

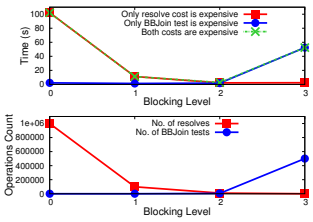

Figure 21: $\mathfrak{R}()$ vs. BBJoin

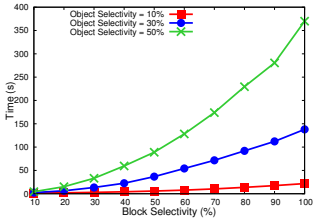

Figure 22: Object Selectivity

8.4 Synthetic Dataset Experiments

To evaluate our approach in a wider range of various scenarios, we built a synthetic dataset generator that allows us to generate datasets with different characteristics. In each synthetic dataset, we control the following input parameters: (i) n: the number of entity-sets, (ii) r: the number of rows in each entity-set, (iii) ℓ: blocking level which leads to the creation of b non-overlapping blocks with size equal to s, (iv) $\mathbb{S}_{\mathcal{B}}(\varphi_i)$: the block selectivity of predicate φ_i, and (v) $\mathbb{S}_{\mathcal{O}}(\varphi_i)$: the object selectivity of predicate φ_i.

In addition, we control the costs of *resolve*, *BBJoin*, *BOJoin*, and *OOJoin* to study their impacts more precisely.

Experiment 5 (Resolves vs. BBJoin Tests). This test employs query (SELECT * FROM R, S WHERE $R.a_i = S.a_i$) to study the trade-off between the resolves and BBJoin tests on LQS, when varying parameter ℓ while fixing the other two parameters to: $n = 2$ and $r = 1,000$. Note that the block/object selectivities do not affect the outcome of this test as we only have one join predicate and every block/object from R joins with exactly one block/object from S.

When $\ell = 0$, which indicates the loosest blocking level, we obtain one block ($b = 1$) with 1,000 rows in it ($s = 1,000$). Clearly, in this case, calling resolve is the bottleneck of LQS since it invokes 999,000 resolves versus performing one *BBJoin* test, see Figure 21. This explains why LQS takes almost 100 seconds when resolve is expensive (*cost*(*resolve*) = 0.1 ms) and almost one second when it is cheap (*cost*(*resolve*) $\approx$ 0 ms). As depicted in Figure 21, when $\ell = 1$ and $\ell = 2$, the number of resolves is comparable to the number of *BBJoin* tests; and hence, the performance of LQS is convergent when resolves are expensive and *BBJoin* tests are cheap and vice versa. Finally, when $\ell = 3$ (viz., the tightest blocking level) we obtain $b = 1,000$ and $s = 1$. LQS invokes 0 calls to resolve versus it performs 500,500 block-to-block join tests. Herein, the cost of *BBJoin* tests is the dominant factor of the overall execution time. This demonstrates why our solution takes almost 1 second when a *BBJoin* test is cheap (*cost*(*BBJoin*) $\approx$ 0 ms) and almost 52 seconds when it is expensive (*cost*(*BBJoin*) = 0.1 ms).

Experiment 6 (Effects of Object Selectivity). Figure 22 uses query (SELECT * FROM R, S WHERE $R.a_i = S.a_i$ AND $R.a_j \geq t$ AND $S.a_k \geq t$) to evaluate the the object selectivity effects on LQS. The previous query contains three predicates: $\varphi_1 : R.a_i = S.a_i$, $\varphi_2 : R.a_j \geq t$, and $\varphi_3 : S.a_k \geq t$.

In this experiment, we create a dataset $\mathcal{D} = \{R, S\}$ by fixing the input parameters to: $n = 2$ and $r = 100,000$, and $\ell = 3$ (which leads to $b = 1,000$ and $s = 100$). Note that, in dataset $\mathcal{D}$, every block/object from R joins with exactly one block/object from S. We vary the block selectivities $\mathbb{S}_{\mathcal{B}^R}(\varphi_2) = \mathbb{S}_{\mathcal{B}^S}(\varphi_3)$ from 10% to 100% (x-axis). We further set three different values for their object selectivities: $\mathbb{S}_{\mathcal{O}^R}(\varphi_2) = \mathbb{S}_{\mathcal{O}^S}(\varphi_3) = 10\%, 30\%$, or 50%.

In addition, we set the costs of *resolve*, *BBJoin*, *BOJoin*, and *OOJoin* to be the same. Due to this assignment, the cost of *OOJoin* overshadows all other costs since the number of

OOJoin operations (performed by the algorithm) surpasses all other operations (i.e., *resolve*, *BBJoin*, and *BOJoin*).

As expected, the higher the selectivity of the query (e.g., $\mathbb{S}_{\mathcal{B}^R}(\varphi_2) = \mathbb{S}_{\mathcal{B}^S}(\varphi_3) = \mathbb{S}_{\mathcal{O}^R}(\varphi_2) = \mathbb{S}_{\mathcal{O}^S}(\varphi_3) = 10\%$) the faster `LQS` is able to evaluate the query and return an answer.

9. RELATED WORK

Entity resolution is a well-recognized problem that has received significant attention in the literature over the past few decades, e.g. [6, 10]. A thorough overview of the existing work in this area can be found in surveys [11, 20]. The majority of previous ER research has focused on improving either its efficiency [16, 21] or quality [5, 7]. With the increasing demand of (near) real-time analytical applications, recent research has begun to consider new ER approaches like analysis-aware ER, progressive ER, incremental ER, etc.

Analysis-aware ER. The work on analysis-aware ER has been proposed in [4, 8, 17, 22, 24], of which [4, 24] are the most related to our work. The `QDA` approach of [4] aims to reduce the number of cleaning steps that are necessary to exactly answer *selection* queries. It works as follows: given a block B, and a complex selection predicate P, `QDA` analyzes which entity pairs do not need to be resolved to identify all entities in B that satisfy P. To do so, it models entities in B as a graph and resolves edges belonging to cliques that may change the query answer. To support a selection query, `QDA` performs *vestigiality analysis* on each block individually to reduce cleaning steps. `QDA` is *not* designed for the larger class of SPJ queries, which is the context of this paper. In contrast, `QuERy` explores a systematic cost-based approach to jointly optimize both cleaning and query processing over dirty data. It exploits pruning due to both selection and join predicates. It only dictates when a block should be cleaned and is agnostic to *how* the block is actually cleaned. Thus, it could exploit vestigiality analysis from `QDA` at the block level to reduce the number of entity pairs that are resolved within a block. In addition, reference [24] is designed to answer *aggregate* numerical queries over large datasets that cannot be fully cleaned. It focuses on cleaning only a sample of data and utilizing that sample to provide "approximate" answers to aggregate queries. It does not prune cleaning steps due to query predicates. However, `QuERy` deals with "exact" answers to SPJ queries based on cleaning only the necessary parts of data needed to answer the query. Moreover, not only are the two approaches designed for different types of queries, their motivation is also very different. While [24] is targeting aggregation over very large datasets, `QuERy` targets applications that perform (near) real-time analysis over dynamic dirty datasets found on the Web.

New ER Approaches. Several approaches, e.g., [3, 26], are considering how to clean the data progressively, while interactively analyzing the partially cleaned data to compute better results. Moreover, there has been various incremental cleaning techniques [12, 25]. Such techniques address the problem of maintaining an up-to-date ER result when data updates arrive quickly.

In addition to such approaches, the ER research community is exploring other novel directions. For example, Data Tamer [23], is an end-to-end data curation system that entails machine learning algorithms with human input to perform schema integration and entity resolution. In addition, NADEEF [9] is a general-purpose data cleaning and repair system that provides appropriate programming abstractions for users to specify data cleaning transformations. The focus of `QuERy` is thus, complementary (though different) to that of [23] and [9]. In fact, we envision that `QuERy` could be useful to these systems to expand their scope to target (near) real-time analysis-aware applications of diverse data sources found on the Web.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have studied the problem of analysis-aware data cleaning. We have developed `QuERy`, a novel architecture for integrating ER with query processing to answer complex `SQL`-like queries issued on top of dirty data. We empirically showed how our approach is significantly better compared to cleaning the entire dataset, especially when the query is very selective.

This research opens several directions for future investigation. Embedding `QuERy` into a DBMS (e.g., PostgreSQL, Spark, etc.) is an interesting direction for future work. Another direction is to extend `QuERy` to deal with other types of queries, e.g., aggregation queries, top-k queries, etc.

11. REFERENCES

[1] http://www.ics.uci.edu/~haltwaij/QuERy.pdf.
[2] http://www.trifacta.com.
[3] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *VLDB*, 2014.
[4] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *VLDB*, 2013.
[5] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.
[6] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 2009.
[7] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 2007.
[8] I. Bhattacharya and L. Getoor. Query-time entity resolution. *JAIR*, 2007.
[9] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, 2013.
[10] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.
[11] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 2007.
[12] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental record linkage. *VLDB*, 2014.
[13] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Record*, 1999.
[14] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *VLDB*, 2009.
[15] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation.
[16] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Record*, 1995.
[17] E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *VLDB*, 2010.
[18] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI*, 2011.
[19] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *ICDE*, 2012.
[20] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *DKE*, 2010.
[21] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, 2000.
[22] Y. Sismanis, L. Wang, A. Fuxman, P. J. Haas, and B. Reinwald. Resolution-aware query answering for business intelligence. In *ICDE*, 2009.
[23] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.
[24] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, 2014.
[25] S. E. Whang and H. Garcia-Molina. Incremental entity resolution on rules and data. *VLDB J.*, 2014.
[26] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *TKDE*, 2013.