Arnold: Declarative Crowd-Machine Data Integration

Shawn R. Jeffery
Groupon, Inc.
sjeffery@groupon.com

Liwen Sun
UC Berkeley
liwen@cs.berkeley.edu

Matt DeLand
Groupon, Inc.
mdeland@groupon.com

Nick Pendar
Groupon, Inc.
npendar@groupon.com

Rick Barber
Stanford University
barber5@cs.stanford.edu

Andrew Galdi
Stanford University
agaldi@stanford.edu

ABSTRACT

The availability of rich data from sources such as the World Wide Web, social media, and sensor streams is giving rise to a range of applications that rely on a clean, consistent, and integrated database built over these sources. Human input, or crowd-sourcing, is an effective tool to help produce such high-quality data. It is infeasible, however, to involve humans at every step of the data cleaning process for all data. We have developed a declarative approach to data cleaning and integration that balances when and where to apply crowd-sourcing and machine computation using a new type of data independence that we term Labor Independence. Labor Independence divides the logical operations that should be performed on each record from the physical implementations of those operations. Using this layer of independence, the data cleaning process can choose the physical operator for each logical operation that yields the highest quality for the lowest cost. We introduce Arnold, a data cleaning and integration architecture that utilizes Labor Independence to efficiently clean and integrate large amounts of data.

1. INTRODUCTION

Our world is becoming increasingly data-driven; vast amounts of data provide unprecedented insights into our own lives [6], our connections to others [4], and the world around us [3]. These data, however, suffer from many data quality issues: they are incorrect, incomplete, and duplicated [24]. Thus, in order to leverage these data in applications, they need to be cleaned and integrated to create a correct, complete, and comprehensive database. Furthermore, applications that use these data tend to require a high-level of data quality.

Achieving such data quality, however, is a challenge as there is typically a long-tail of "corner cases": quality issues in the data that do not easily fit into any pattern, and as such are not easily handled by existing "good-enough" data cleaning solutions. For instance, state-of-the-art entity resolution algorithms have many challenges in determining duplicates in real-world scenarios [15]. Since humans excel at finding and correcting errors in data, manual input, or *crowd-sourcing*, can be used to provide the high quality data required by these applications.

ID	Name	Address	Category
1	SFO	San Francisco, CA	Airport
2	Washington Park	Burlingame, CA	Park
3	French Laundry	Yountville, CA	———

Table 1: An example geographic location database

As a motivating example, consider a data integration system designed to produce a database of geographic locations such as businesses, parks, and airports. Such a database is necessary to drive many location-based services. Input data for this database may come from a variety of sources, including licensed datasets [5], governmental sources [26], and user-generated content. Each data source needs to be individually cleaned then integrated to create a single database. A simplified version of such a database is shown in Table 1 that consists of a set of records, each of which describes a physical location containing a unique identifier, name, address, and category. The quality requirements for these databases are high; for instance, an incorrect address could cause a user of the database to go to the wrong place. Figure 1 shows a simplified cleaning pipeline necessary to build such a database. First, the address for each record is cleaned to correct for spelling errors and formatted in a standard manner. Second, each record is classified to determine the category of the location. Finally, each input record is deduplicated with existing records in the database.

Figure 1: An example cleaning pipeline to produce a location database. Each node represents a cleaning operation that the system applies to each input record.

In this example, the crowd can be a powerful tool to help produce a high-quality database. For instance, a classification algorithm may categorize French Laundry as a laundromat based on the name; however, a human would easily discover that French Laundry is in fact a restaurant.

Of course, human involvement does not easily scale; it is

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2013.
6th Biennial Conference on Innovative Data Systems Research (CIDR '13)
January 6-9, 2013, Asilomar, California, USA.

expensive to involve humans in the data cleaning process. Given that it is not feasible to consult the crowd for every operation and for every record, one must decide when and how to utilize crowd-sourcing. In a typical data integration system, the complexity of such a decision is compounded by a multitude of factors.

Heterogeneous data quality. Since data are integrated from heterogeneous sources, they arrive with various quality levels. For the *good* records, it is often sufficient to process them with automated algorithms; they would only benefit to a small degree from crowd-sourcing since the quality is already high. The *bad* records are typically not easily handled by automated mechanisms and need to be curated by a human. For example, an address cleaning algorithm will fail if the address is very dirty and cannot be parsed.

Non-uniform quality requirements. There is typically a wide variance in the importance to the application of the records and attributes in the database. For example, the address attribute is critical to power location-based applications, while the category attribute is merely a helpful guide. Similarly, SFO (San Francisco Airport) is a popular location in the San Francisco Bay Area, while Washington Park is only known to a small community. For these more important records or attributes, applications desire high-quality data and thus more resources should be invested in cleaning them.

Multiple cleaning operations. A typical data integration system consists of tens to hundreds of cleaning operations. For each task, the system must decide how and when to use human input.

Different expertise. Human and machines excel at different tasks. Machine algorithms are good at executing routine tasks efficiently and scalably. Humans, on the other hand, are able to utilize external data or infer patterns through human judgement that may not be evident in the data.

Different costs. Machines, in general, are inexpensive, while human involvement is expensive. Furthermore, the response time of machine algorithms is generally much faster than that of the crowd. However, the converse may be true. For some complex machine learning tasks, it may take large amounts of computing resources and time to train and execute a model while a human could perform the task easily and cheaply.

Multiple crowds. While generally crowd-sourcing has referred to a single amorphous crowd, in reality there are typically many different crowds. Basic crowds range from the general-purpose crowds (e.g., Mechanical Turk [20]) to more full-featured crowd platforms [2]. For specialized tasks, crowds can be highly-trained workers, such as an internal editorial staff, or groups designed for certain purposes (e.g., physical data collection [10]). Each of these crowds has a different level and type of expertise. Workers with domain knowledge of data (e.g., an editorial staff) are more expensive to use but can provide better results. The system must decide between different crowds and expertise.

In general, applications desire a high-quality, integrated database with minimal cost; they do not want to understand the complex trade-offs presented by each of the factors above. To this end, we have developed a declarative approach to data cleaning and integration that balances when and where to apply crowd-sourcing and machine computation. The primary mechanism we use to drive our system is a new type of data independence we term *Labor Independence*. Labor Independence provides a high-level declarative interface that allows applications to define the quality and cost requirements of the database, without needing to specify how that database is produced.

While declarative approaches to crowd-sourcing have been extensively studied in the literature (e.g., [8, 19, 23]), Labor Independence is the first approach that hides *all* details of the underlying crowd-sourcing mechanisms from the application. Existing approaches apply declarative principles at the operator, task, or crowd level, but still force the application to specify some portion of the crowd-based execution. For example, in order to use CrowdDB's crowd-sourcing functionality, an application must speficy crowd-sourcing operations in the DDL and DML [8], requiring intimate knowledge of the operations that the crowd is capable of perfoming and their efficacy. When utilizing crowd-sourcing at large scale, however, such knowledge is largely inaccessible to an application developer. In contrast, Labor Independence provides to applications an interface in which they only need to specify the data they want, the quality they desire, and how much they are willing to pay, without any knowledge of the crowd-sourcing mechanisms that produced those data. In fact, such a system could not utilize crowd-sourcing at all as long as it met the application's desires, all without intervention from the application developer. As a result, systems can seamlessly integrate crowd-sourcing without having to understand the details of the crowd(s).

Beneath that interface, Labor Independence abstracts data cleaning tasks such that they can be performed by machines or the crowd. It does this by dividing logical operations that should be performed on each record from the physical implementations of that operation. Using this layer of independence, the data integration system can choose the physical operator for each logical operation that yields the highest quality for the lowest cost.

In order to realize Labor Independence, we define a unified mechanism for expressing and tracking quality and cost in a data integration system. Our quality model is based on the intuition that the closer the database matches the reality it represents, the better quality it is. Using this intuition, we derive a multi-granularity quality model that quantifies the closeness to reality at the attribute, record, and database levels. We also define a cost function that distills all costs, including time and resources, into a single monetary cost.

We present Arnold, a system that utilizes Labor Independence to clean and integrate data in an efficient manner. Arnold processes streams of input records from multiple sources through set of logical operations, each implemented by one or more physical operators. In order to drive its optimization, Arnold continuously estimates the quality of the data it is processing as well as the quality and cost of its physical operators. Using these estimates, Arnold optimizes the selection of physical operators for each record and operation to maximize quality or minimize cost.

This paper is organized as follows. In Section 2 we present Labor Independence. We provide an overview of Labor Independence's quality and cost models in Section 3. In Section 4 we outline the Arnold Architecture that instantiates Labor Independence. Section 5 describes the optimization algorithms that enable Arnold to efficiently blend machine and human labor. Finally, we highlight related work in Section 6 and conclude in Section 7.

2. LABOR INDEPENDENCE

In this section we describe Labor Independence, the fundamental mechanism we use to seamlessly blend machine and human labor.

In this work, we consider large-scale, pay-as-you-go data cleaning and integration systems [7, 17]. In such systems, streams of input records are cleaned and integrated into a single database using multiple cleaning operations. These cleaning operations are expressed as a pipeline of potentially hundreds of fine-grained operations that successively process input records to produce an integrated database. Such a database is a constantly evolving collection of records, each of which is represented by a set of attribute-value pairs. The number of sources in these scenarios typically number in the hundreds or thousands and have high variance in correctness, completeness, and sparsity. In total, such a system may process billions of records. A prerequisite for any data-driven application built on top of these systems is a high-quality, integrated database.

2.1 Overview

Here we present a new type of data independence, *Labor Independence*, that provides applications with a high-level declarative interface to specify the quality requirements of the data and the cost the application is willing to pay. Beneath that interface, Labor Independence separates logical cleaning operations from the physical mechanisms that perform those tasks. A system implementing Labor Independence has a set of one or more physical operators for each logical operation, each of which has different cost and quality. Utilizing this layer of independence, the system is then free to choose, for each logical operation and for each record, the appropriate physical operator.

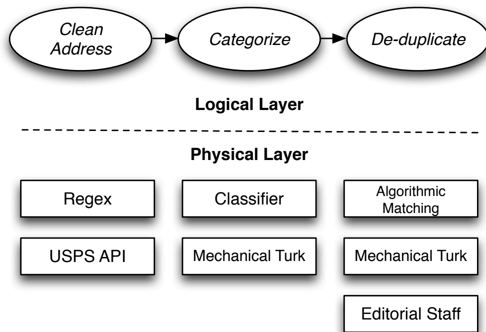

Figure 2: The logical and physical layers of Labor Independence for a location data cleaning pipeline.

We illustrate Labor Independence in Figure 2 as applied to the location database example. In this example, we represent each of the cleaning tasks (*Clean Address*, *Categorize*, and *De-duplicate*) as a logical operation; that is, it specifies the action that should occur for each record, not how to perform that action. For each of these operations, there is one or more physical operators. For example, the *Categorize* logical operation could be carried out by either an algorithmic classifier or by asking the crowd. Note that there may be multiple algorithmic or crowd-based operators for a single logical operation. For example, *Clean Address* could be carried out using a regular expression or by utilizing an external service [29], whereas de-duplication could be done either algorithmically or by two different crowd-based mechanisms.

We now describe the fundamental components of Labor Independence in more detail.

2.2 Declarative Interface

An application interacts with a system implementing Labor Independence through a declarative interface using two parameters, *quality* and *cost*. Quality is quantified as a numerical score in $[0, 1]$. Cost is a single number that unifies all types of costs, including money, time, and resources, into a monetary cost. In Section 3, we will discuss the quality and cost models in detail. Using the quality and cost parameters, an application can specify the level of quality it needs and the amount of money it is willing to pay. In some instantiations of Labor Independence, an application may only specify one of the two parameters, in which case the system will minimize or maximize the other. For example, a user may request that the system maximize the quality of the data utilizing less than $1.00 per record.

2.3 Logical Layer

The primary abstraction at the logical layer is a *logical operation*, which defines a semantic operation that is to be performed on an input record. Each logical operation is specified via the input, output, and the set attributes of a record can be mutated by this operation. For example, the *CleanAddress* operation takes as input a record, outputs a record, and mutates the address attribute but not others. Importantly, a logical operation does not specify how that operation is to be implemented.

Logical operations are combined into a *logical operation graph* that defines the order in which logical operations should be applied to input records. This ordering may be a fixed ordering of logical operations, or a dependency graph where each logical operation depends on zero or more other logical operations.

2.4 Physical Layer

At the physical layer, each logical operation has one or more corresponding *physical operators*: a concrete implementation that specifies how to carry out the logical operation. A physical operator may utilize algorithmic mechanisms or crowd-sourcing techniques, but it must conform to the specifications of the logical operation that it implements. Each physical operator has an associated quality and cost: the expected quality of data produced by the operator and the monetary cost of executing it.

3. QUALITY AND COST

In order to make informed decisions about when and where to apply manual input, Labor Independence depends an understanding of the *quality* and *cost* of the records it processes. In this section we detail our quality and cost models.

3.1 Quality Model

The intuition underlying our quality model is that applications need data that closely represents reality; thus, we quantify data quality by the degree to which the database matches the reality it represents. We describe the salient features of our quality model here and defer a detailed discussion to future work.

The primary building block of our quality model is a *dissonance function*, that measures the difference between an attribute value and the reality it represents.

DEFINITION 1 (DISSONANCE FUNCTION). *Let r be a record in our database and $r.a$ be an attribute a of record r. A dissonance function for attribute a, denoted as z_a, takes as input the record r and the true value of $r.a$, $T(r.a)$, and returns a numerical score $z_a \in [0, 1]$ that quantifies the distance between $r.a$ and $T(r.a)$.*

The smaller the value for z_a, the closer the value $r.a$ is to reality. If $r.a$ has a missing value, then $z_a = 1$. For each attribute a in an application, the system must define a dissonance function. The implementation of these dissonance functions are application-specific. For example, we can define dissonance functions for the example location database: z_{name} is the normalized string edit distance between the name string and the true name string, and $z_{address}$ is the normalized euclidean distance between the address and the true address.

We assert that defining a dissonance function for every attribute is a tractable problem because the number of attributes is at human scale for any given application (e.g., in the tens to hundreds). Furthermore, since a dissonance function is at the granularity of a single attribute value, it is easy for a human to reason about and derive a dissonance function.

In order to compute a dissonance function, we need to know the true value for the attribute $T(r.a)$. Thus, we rely on an *oracle* that is capable of providing the value of $T(r.a)$. An oracle may be a trained domain expert or a sophisticated crowd-sourcing algorithm designed to attain high-fidelity answers (e.g., [25]). Given $r.a$ and the oracle-provided $T(r.a)$, computing a dissonance function is trivial (e.g., compute the edit distance between $r.a$ and $T(r.a)$).

Using the dissonance value z_a, we define the quality of an attribute value a as:

$$q_a = 1 - z_a$$

Furthermore, using the quality definition for a single attribute value, we can derive aggregate quality scores for coarser-granularity objects such as an entire record and the entire database by simply using an average of the single quality scores. For instance, the quality of a record is the average of its individual attribute qualities, and the quality of the entire database is the average quality of all records.

Of course, querying the oracle to discover the true value of an attribute is expensive. For example, if we want to assess the average quality of an attribute (e.g., address) in the entire database, we cannot afford to send all the records to the oracle. Instead, we use sampling estimation. First, we draw some uniform samples from the database. For each record r in the sample, we send it to the oracle that provides the true value of its address $T(r.address)$. We then compute a quality score for $r.address$ using the appropriate dissonance function. Given the quality scores of all the samples, we use their average to infer the average quality of the entire population. The inaccuracy of this estimation is be bounded by standard error [16].

3.1.1 Missing and Duplicate Records

In any data integration system, both missing and duplicate records are a common challenge. We can extend our quality model to capture both missing and duplicate records at the aggregate-level. Conceptually, a missing record has a quality of 0. Of course, it is unknown what records are missing from a database. Therefore, we can use standard population estimation techniques or even crowd-based estimation [28] to estimate the number of missing records.

Similarly, for duplicates we assign a *uniqueness score* to a set of records corresponding to the percentage of duplicates in that set. The set may be the entire database or subsets of the records (e.g., the records in each postal code). Uniqueness scores can also be estimated using sampling techniques (e.g., [12]).

3.1.2 Weighted Quality

In most data-driven applications, there is a wide variance in the *importance* of different attributes and records: some data is queried more frequently, more valuable to a business, or drives critical application features. This observation can help realize further efficiency in our data cleaning efforts by focusing on the important data. To quantify importance, we define *importance weights* at both the attribute and record level. We define for each attribute and record in an application an importance weight in $[0, 1]$ that denotes the relative importance of the attribute or record to the application. Since there are typically a large number of records, record weights can be set at an aggregate level. For instance, in the location database example record weights can be set based on postal code. Importance weights are normalized across all attributes or records. We use these weights to compute a weighted average of quality at the record and database level.

3.1.3 Quality of a Physical Operator

In order to choose physical operators based on quality, a system implementing Labor Independence needs to understand the quality of each physical operator. We define the quality of a physical operator as the average quality score of its output records. By definition, this quality is a numerical score in $[0, 1]$. We discuss in the next section how we track quality for physical operators.

3.2 Cost Model

Similar to quality, Labor Independence needs a unified mechanism for tracking the cost of each record as well as each physical operator. Costs can come in many forms: time for an operation to complete, monetary cost for hardware to run algorithms, and payment to crowd workers. We reduce all of these costs to a single monetary cost.

3.2.1 Cost of a Physical Operator

Each physical operator incurs a cost when it is executed. The primary challenge in tracking the cost of physical operators is that the cost may not be known *a priori*, especially for crowd-based operators. For example, the completion time of finding missing data can vary widely. As described in the next section, we monitor the cost of each operator and use average cost to predict future costs.

4. THE ARNOLD ARCHITECTURE

In this section, we describe the Arnold Architecture that instantiates Labor Independence to provide holistic crowd-machine data integration. Figure 3 shows the components of Arnold.

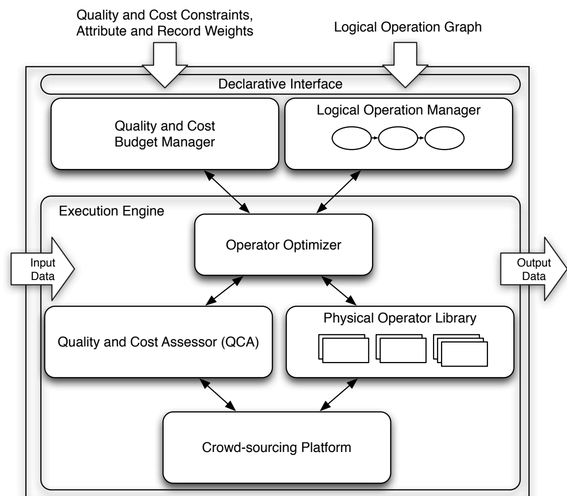

Figure 3: The Arnold Architecture

Declarative Interface. A user interacts with Arnold through a declarative interface. As per Labor Independence, the user specifies cost and quality constraints for each record processed by Arnold. Additionally, a user can define importance weights for attributes or records. If unspecified, every attribute and record is treated uniformly.

The user also specifies the logical operations that should occur for each record and in what order. The domain of logical operations is defined as part of system set-up, as well as the corresponding set of physical operators (stored in the *Physical Operator Library*).

These specifications are stored in a *Budget Manager* and a *Logical Operation Manager*, respectively, that are consulted as the system runs to ensure it is meeting the user's requirements.

Execution Engine. This component is responsible for processing records given the user's declaration. We describe the operation of the Execution Engine in Algorithm 1. As records stream in from input sources, they are placed on a processing queue. The Execution Engine consumes records from this queue and consults the *Operator Optimizer* to determine the next physical operator for each record. After a record is processed by a physical operator, the Execution Engine passes it to the *Quality and Cost Assessor* module to assess the quality of the record. Finally, the engine asks the Budget Manager if the record has met the user's cost and quality constraints. If so (or if the record has been processed by all logical operations), the engine sends the record to the output data stream. If the record does not meet the constraints, the engine enqueues the record back on to the queue for further processing.

Operator Optimizer. This component determines the appropriate next physical operator to process a record. After checking which logical operations have already been applied to the record, the optimizer consults the Logical Operation Manager to determine which are the possible next logical operations. It then runs an optimization algorithm given the set of physical operators available for the possible logical operations as well as the the current and desired quality and cost of the record. We discuss the optimization algorithms

Algorithm 1 Execution Engine's algorithm for processing records

Precondition: input records enqueued into $processQueue$
1: **while** "*monkey*" $\neq$ "*banana*" **do**
2: $record \leftarrow processQueue.dequeue()$
3: $op \leftarrow operatorOptimizer.bestOperator(record)$
4: $record \leftarrow op.process(record)$
5: $qca.updateQualityAndCost(record)$
6: **if** $budgetManager.meetsConstraints(record)$ **then**
7: $outputQueue.enqueue(record)$
8: **else**
9: $processQueue.enqueue(record)$
10: **end if**
11: **end while**

used by the Operator Optimizer in Section 5.

Operator	Attribute	Quality
Regex Cleanup	Address	0.73 (± 0.10)
USPS API	Address	0.87 (± 0.05)
Category Classifier	Category	0.81 (± 0.07)
M. Turk Classifier	Category	0.92 (± 0.02)
...	...	...

Table 2: Example quality estimates tracked by the QCA

Quality and Cost Assessor (QCA). This module is responsible for continuously tracking the quality and cost of each record. As described in Algorithm 1, the QCA receives records after they have been processed by each operator (and the actual cost to process that record by that operator, if known). It is then responsible for assessing and updating the quality and cost for that record. To do so, the QCA tracks, for every operator and for every attribute, an estimate of the quality (and error bound) that can be expected from that operator for that attribute. It continuously monitors the quality of the record stream output by the operator using *reservoir sampling* [30].

These estimates are stored in a quality estimate table. For example, Table 2 shows a truncated estimate table for the location database scenario. For each physical operator from Figure 2 and for each attribute in the application (e.g., Name, Address, Category), there is an entry in the table with the estimated quality that that operator can produce for that attribute. Using the estimation, the QCA updates the quality of each of the attributes in the record given the previous operator. Of course, if a record was selected as a sample to learn the true quality, it will be assigned that quality.

The QCA is also responsible for tracking the cost of processing each record by each operator. Similar to quality tracking, the QCA utilizes the cost estimate or the exact cost from the operator to update the record's remaining budget.

Crowd-Sourcing Platform. Crowd-sourcing may be utilized by many physical operators as well as by the QCA. Thus, Arnold includes a crowd-sourcing platform that interfaces with one or more crowds and hides the details of the crowd from the rest of the system. The details of this platform are outside the scope of this paper.

5. ADAPTIVE OPERATOR OPTIMIZATION

Having outlined Arnold's overall functionality, we now address how Arnold optimizes the set of operators to apply to a given record and in what order.

There are many questions to consider when approaching this problem. We first outline the scope of design considerations, and then propose a solution to one version of the problem.

Choice of Objective Function. There are a range of objective functions for which the system can be designed to optimize. On one hand, the system can be given a cost budget (a strict limit) and try to maximize the quality of data it produces. Conversely, it could be given a quality threshold it must meet and then try to minimize cost. For ease of exposition, we focus on maximizing quality given a fixed budget, but these two problems are essentially duals of each other and our solutions are equally effective for both.

Fixed vs. Dynamic Budget. As discussed above, each record gets a fixed budget upon entering the system and each operator subtracts from that budget. This budget could be set once at the start of processing, or it could be dynamically adjusted as the system processes the record. Such an adjustment could occur as the system gains a deeper understanding of the data in the record. For example, if after cleaning a record's address the system learns that the record refers to a location in a popular area (e.g., downtown San Francisco), then the system may increase that record's budget.

Ordering of Operations. We have presented the logical operator graph as a set of logical operations occurring in a fixed order, e.g., *Clean Address* followed by *Categorize* followed by *De-duplicate*. An alternative is to choose any logical operator at any time, given that a set of prerequisites is met. For example, the system can apply *Clean Address* or *Categorize* in either order, but *Clean Address* is a prerequisite for *De-duplicate*, which relies on accurate addresses. If we allow alternate orderings, it transforms the linear graph into a directed acyclic graph. Working with this additional complexity guarantees at least as good and possibly better quality, but the resulting optimization problem becomes more difficult.

Repetition or Omission of Logical Operators. So far, we have discussed the system applying each logical operation once and only once to each record. However, it may be beneficial to allow the system to re-execute some logical operations with costlier physical operators. For example, if a record that had its address cleaned using a low-cost and low-quality operator was later discovered to have high importance, it may be desirable to re-clean its address using a higher-quality address cleaning operator. Conversely, some operators may not be necessary for some records and can be omitted to save costs. For example, if a record comes from a source that is known to have high-quality addresses, the address cleanup can be skipped.

Different choices for these design considerations raise different questions as to how to allow a record to move through the graph. We now formalize one variant of the problem and present a solution. Other variants of the problem will be addressed in future research.

5.1 Optimization on a Linear Operator Graph

We focus on a linear operator graph, where operators are statically ordered in a path. Thus, the primary challenge is to chose which physical operator to employ for each logical operation. In order to solve this problem, we create a combined graph with each logical operation represented as a node in the graph and each physical operator represented as a directed edge connecting two nodes. An example of this graph is depicted in Figure 4. The problem now becomes one of dynamically choosing the best path through this graph for any given record that maximizes quality subject to budget constraints.

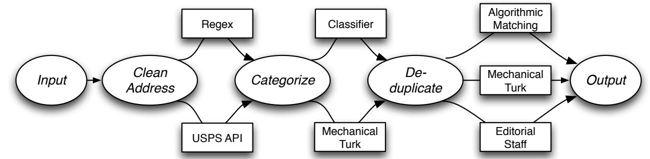

Figure 4: Visualization of a linear operator graph

The simplest solution to the problem would be to enumerate all possible paths with their associated cost and quality, and at any given point in the graph, choose the path which maximizes quality according to the currently available budget. This method, though effective for small graphs, will quickly become unreasonable as more operators are added. Next, we present a more efficient solution based on dynamic programming.

5.1.1 The Optimization Problem

We formulate this problem as an optimization problem where the input is a linear operator graph G, record R, and budget b; and the objective is to choose a path P that passes through every node of G according to

$$\begin{aligned} \underset{P}{\text{maximize}} \quad & Quality(P) \\ \text{subject to} \quad & Cost(P) \leq b \end{aligned}$$

The resulting optimization problem requires us to be able to choose the best bath from any node given any budget. If we assume that the per record budget is an integer that is not very large and has a known maximum value B, the problem can be efficiently solved offline using dynamic programming. Working backwards from the output node of the graph, at each node and for any given budget, choose the highest quality path that can be afforded. If n is the number of nodes, the problem can be solved in $O(nB)$ time and space. Note that this algorithm runs only once and materializes an optimal path for each possible budget value at every step. When a record with a budget b enters the system, we simply lookup the optimal path for budget b. If the budget changes during execution of the chosen path, an additional lookup is needed to pick the optimal path moving forward.

The design choices described above conform to the most common practical requirements that we have seen in the real-world. More flexibility of operator selection may lead to better results, but increases the complexity of the optimization. For example, if a record can go through the logical operators in arbitrary order, the optimization problem requires a $O(2^n B)$ dynamic programming algorithm. We will explore such complexity in future work.

6. RELATED WORK

There has been much recent interest in crowd-sourcing applied to data management and data cleaning. At the systems-level, many groups propose a declarative approach to utilizing crowd-sourcing in a database [8, 19, 23]. Labor Independence and Arnold are closely related to these systems; similar to these systems, Labor Independence hides the details of where and how crowd-sourcing is used. However, a primary distinction is that the declarative interface of Labor Independence is at the *application-level* rather than at the operator-level; that is, applications specify the level of quality they want for their data overall (and the price they are willing to pay), rather than specifying constraints at the individual operator level. On the other hand, the granularity of optimization in Arnold is on a per-record basis, similar to Eddies [1] for optimizing relational queries. Arnold is focused specifically on providing high-quality data cleaning instead performance optimization of general data processing scenarios.

At a more detailed level, there is a wealth of work on how to use the crowd to accomplish individual tasks including filtering [22], graph search [21], max discovery [11], sorting and joining [18], and entity resolution [31]. These approaches could be incorporated into the Arnold architecture as physical operators. Roomba [14] is a precursor to our work in that it is focused on optimizing how and when to apply human involvement in an individual task; Arnold extends that work to optimize human input across multiple tasks.

Data cleaning and integration is a well-studied problem [13, 24]. The work closest to ours is the AJAX system [9], which separates the logical and physical operations for data cleaning. Due to the incorporation of crowdsourcing, our work is different than AJAX in several ways. First, we incorporate both quality and cost as a first-class elements of our model and define our declarative interface using these parameters. Second, we consider a much broader range of logical and physical operations, which can be algorithmic or crowd-based. Finally, our optimization problem involves a multitude of new factors that were not present in AJAX, such as the trade-off between crowds and machine algorithms.

7. CONCLUSION

As applications come to depend on data to a greater degree, so too does their expectation of the quality of those data. While traditional data cleaning and integration systems are capable of producing "good" data, human input, or crowd-sourcing, is required in order to deal with the long-tail of quality issues in large-scale data integration systems. However, human involvement is expensive, and the choice of how utilize humans is non-trivial. Fundamentally, a data-driven application desires high-quality data, and does not want to be concerned with the complexities of crowd-sourcing.

In this work, we developed Labor Independence, a declarative approach to data cleaning and integration that enables applications to specify their quality and cost requirements for their data without needing to understand how that data are produced. We have described Arnold, a system that implements Labor Independence to clean and integrate large numbers of input records while holistically using machine and crowd labor.

Arnold provides a rich test-bed for us to further research the union of machines and humans. In the future, we intend to explore the cost and quality trade-offs provided by Labor Independence in more detail, and develop more sophisticated optimization algorithms. We also look to provide more insight into the real-world data integration challenges we face at Groupon and demonstrate Arnold's effectiveness at addressing these issues.

8. REFERENCES

[1] Avnur, R., and Hellerstein, J. M. Eddies: continuously adaptive query processing. In *Proc. of ACM SIGMOD 2000*, pp. 261–272.

[2] Crowdflower. http://www.crowdflower.com.

[3] Culler, D., Estrin, D., and Srivastava, M. Overview of Sensor Networks. *Computer 37*, 8 (Aug. 2004), 41–49.

[4] Facebook. http://www.facebook.com.

[5] Factual. http://www.factual.com.

[6] Fitbit. http://www.fitbit.com.

[7] Franklin, M., Halevy, A., and Maier, D. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec. 34*, 4 (Dec. 2005), 27–33.

[8] Franklin, M. J., Kossmann, D., Kraska, T., Ramesh, S., and Xin, R. CrowdDB: Answering Queries with Crowdsourcing. In *Proc. of ACM SIGMOD 2011*.

[9] Galhardas, H., Florescu, D., Shasha, D., Simon, E., and Saita, C.-A. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proc. of VLDB 2001*.

[10] Gigwalk. http://gigwalk.com/.

[11] Guo, S., Parameswaran, A., and Garcia-Molina, H. So Who Won? Dynamic Max Discovery with the Crowd. In *Proc. of ACM SIGMOD 2012*, pp. 385–396.

[12] Haas, P. J., Naughton, J. F., Seshadri, S., and Stokes, L. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB* (1995), pp. 311–322.

[13] Hellerstein, J. M. Quantitative Data Cleaning for Large Databases. United Nations Economic Commission for Europe (UNECE), 2008.

[14] Jeffery, S. R., Franklin, M. J., and Halevy, A. Y. Pay-as-you-go User Feedback for Dataspace Systems. In *Proc. of ACM SIGMOD 2008*.

[15] Köpcke, H., Thor, A., and Rahm, E. Evaluation of entity resolution approaches on real-world match problems. *PVLDB 3*, 1 (2010), 484–493.

[16] Lohr, S. *Sampling: Design and Analysis.* Advanced Series. Brooks/Cole, 2009.

[17] Madhavan, J., Jeffery, S., Cohen, S., Dong, X., Ko, D., Yu, G., and Halevy, A. Web-scale Data Integration: You can only afford to Pay As You Go. In *CIDR 2007*.

[18] Marcus, A., Wu, E., Karger, D., Madden, S., and Miller, R. Human-powered Sorts and Joins. *Proc. of VLDB Endow. 5*, 1 (Sept. 2011), 13–24.

[19] Marcus, A., Wu, E., Madden, S., and Miller, R. C. Crowdsourced Databases: Query Processing with People. In *CIDR 2011*.

[20] Mechanical Turk. https://www.mturk.com/.

[21] Parameswaran, A., Sarma, A. D., Garcia-Molina, H., Polyzotis, N., and Widom, J. Human-Assisted Graph Search: It's Okay to Ask Questions. *Proc. of VLDB Endow. 4*, 5 (Feb. 2011), 267–278.

[22] Parameswaran, A. G., Garcia-Molina, H., Park, H., Polyzotis, N., Ramesh, A., and Widom, J. Crowdscreen: Algorithms for Filtering Data with Humans. In *Proc. of ACM SIGMOD 2012*, pp. 361–372.

[23] Park, H., Pang, R., Parameswaran, A., Garcia-Molina, H., Polyzotis, N., and Widom, J. Deco: A System for Declarative Crowdsourcing. In *Proc. of VLDB 2012*.

[24] Rahm, E., and Do, H. H. Data Cleaning: Problems and Current Approaches. *IEEE D. E. Bull. 23*, 4 (2000), 3–13.

[25] Ramesh, A., Parameswaran, A., Garcia-Molina, H., and Polyzotis, N. Identifying Reliable Workers Swiftly. Technical report, Stanford University, 2012.

[26] San Francisco Data. `data.sfgov.org`.

[27] Sarawagi, S., and Bhamidipaty, A. Interactive Deduplication Using Active Learning. In *Proc. of ACM SIGKDD 2002* (New York, NY, USA), pp. 269–278.

[28] Trushkowsky, B., Kraska, T., Franklin, M. J., and Sarkar, P. Getting it all from the crowd. *CoRR abs/1202.2335* (2012).

[29] USPS. `www.usps.com`.

[30] Vitter, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1 (Mar. 1985), 37–57.

[31] Wang, J., Franklin, M., and Kraska, T. CrowdER: Crowdsourcing entity resolution. *PVLDB 5*, 11 (2012).