Generating Example Data for Dataflow Programs

Christopher Olston
Yahoo! Research
2821 Mission College Blvd.
Santa Clara, CA
olston@yahoo-inc.com

Shubham Chopra
Yahoo! Research
2821 Mission College Blvd.
Santa Clara, CA
shubhamc@yahoo-inc.com

Utkarsh Srivastava
Yahoo! Research
2821 Mission College Blvd.
Santa Clara, CA
utkarsh@yahoo-inc.com

ABSTRACT

While developing data-centric programs, users often run (portions of) their programs over real data, to see how they behave and what the output looks like. Doing so makes it easier to formulate, understand and compose programs correctly, compared with examination of program logic alone. For large input data sets, these experimental runs can be time-consuming and inefficient. Unfortunately, sampling the input data does not always work well, because selective operations such as filter and join can lead to empty results over sampled inputs, and unless certain indexes are present there is no way to generate biased samples efficiently. Consequently new methods are needed for generating example input data for data-centric programs.

We focus on an important category of data-centric programs, *dataflow programs*, which are best illustrated by displaying the series of intermediate data tables that occur between each pair of operations. We introduce and study the problem of generating example intermediate data for dataflow programs, in a manner that illustrates the semantics of the operators while keeping the example data small. We identify two major obstacles that impede naive approaches, namely (1) highly selective operators and (2) noninvertible operators, and offer techniques for dealing with these obstacles. Our techniques perform well on real dataflow programs used at Yahoo! for web analytics.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms, Experimentation

1. INTRODUCTION

An increasingly popular data processing paradigm is *dataflow programming*, whereby processing is specified via acyclic graphs. Source nodes denote input data sets, and sink nodes denote output data sets to be generated by running the dataflow program. Intermediate nodes denote set-transformation operations drawn from a suite of operator templates. The operator templates typically resemble relational algebra primitives (e.g., project, filter, join) and/or functional programming primitives (e.g., map, reduce). Recent examples of dataflow programming systems include Aurora [1], Dryad [9], Map-Reduce [7], Pig [13], River [2] and Tioga [15].

As with all programming paradigms, the process of constructing a correct dataflow program is typically an iterative one: The user makes an initial stab at composing a program, submits it to the system for execution, and inspects the output (or failure log) to determine whether the program had the intended effect. If not, the user revises the program and repeats the cycle. Dataflow programs often access large data sets and hence take a long time to execute, so the iterative development process can be very inefficient.

To circumvent this inefficiency, users may choose to create side data sets, consisting of small samples of the original input data sets, for experimentation. Unfortunately this method does not always work well. For example, suppose the program performs an equijoin of two data tables `A(x,y)` and `B(x,z)` on attribute `x`. If the original data contains many distinct values for `x`, then it is unlikely that a small sample of `A` and a small sample of `B` contain any matching `x` values [5]. Hence a program involving a join over the sample data tables may well produce an empty result, even if the program is correct. Similarly, a program with a selective filter executed on a sample data set may produce an empty result.

One may consider biasing the sample so as to ensure nonempty and uniformly sampled join or filter outputs, as suggested in [5]. Unfortunately such methods are only efficient if the appropriate indexes have been created in advance. In the context of ad-hoc programming one cannot always presuppose the availability of indexes that match certain predicates embedded in the program.

Besides, for the purpose of understanding the semantics of a dataflow program, it is not necessary to provide a uniform sample of the program's output. In fact, it is not even necessary to provide a *subset* of the output—if real records are too expensive to obtain, carefully constructed synthetic records can suffice, as we demonstrate next via a simple example.

1.1 Illustrative Example Data

Figure 1 shows an example dataflow program that finds web surfers who tend to visit high-pagerank pages. The program joins two data tables: a log of page visits (`Visits(user, url, time)`) and a catalog of pages and their

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

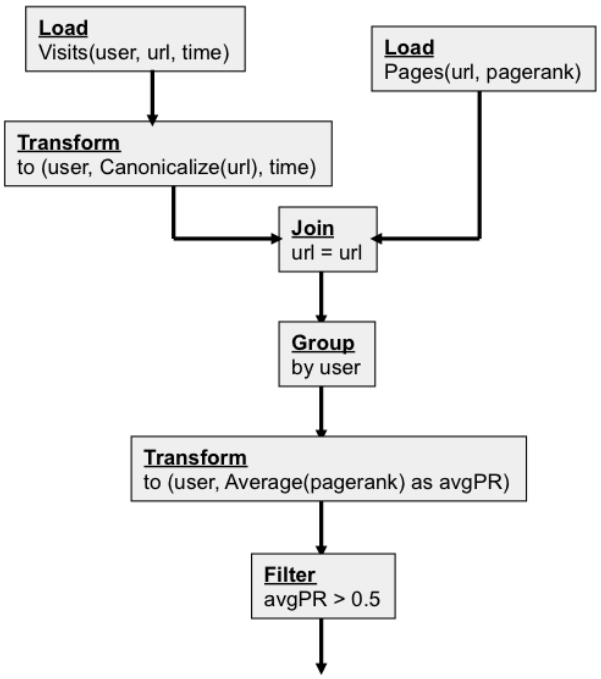

Figure 1: A dataflow program.

pageranks (`Pages(url, pagerank)`), after first running the log entries through a function that converts URLs to a canonical form. After the join, the program groups records by user, computes the average pagerank for each user, and then filters users by average pagerank.

Figure 2 augments Figure 1 with example records that illustrate how data flows through the program's operators. The salient properties are: First, the data seems realistic, i.e., we might expect to find data of this form in the real `Visits` and `Pages` tables. Second, the example data illustrates the key properties of each operator. For example, by inspecting the example data we observe that the GROUP operator forms nested sets that can contain multiple records[1] (e.g., for Amy), and the FILTER operator eliminates Fred, who's average pagerank is too low. Third, the data is concise, i.e., only a few records are shown at each step.

The data in Figure 2 was generated *automatically* by an algorithm we have designed. The capability for automatic generation of realistic and concise example data to illustrate dataflow semantics has significant real-world implications: It enables a new type of tool for helping users construct dataflow programs, in which illustrative example data is displayed on edges between dataflow operators as the user assembles them. A tool of this kind can help users reason about the behavior of their programs as they are composing them, without having to iterate over the full data to see whether they implement the intended semantics. Perhaps of equal importance, such a tool can also help a user understand a dataflow program written by somebody else, e.g., a predecessor who no longer works at the company. Lastly, this type of tool may be helpful in learning a new dataflow language or operator.

[1]In this example, we have used the semantics of the GROUP operator in Pig Latin [13].

1.2 Contributions

In this paper we formalize the problem of generating example data to illustrate the semantics of dataflow graphs, and identify the characteristics that are desirable for high-quality example data. We then present a novel example generation algorithm based on deliberate *nonuniform sampling* combined with *data synthesis*. We show that our algorithm is efficient and produces high-quality output on real dataflow programs, whereas simple baseline approaches perform poorly. Output quality is judged according to three key objectives: realism, conciseness and completeness, which we formalize and quantify.

Our work is motivated by our experience helping users write and debug dataflow programs in the Pig [13] system. While working with users we often find ourselves constructing example intermediate data tables by hand on a sheet of paper, a labor-intensive process that impelled us to seek automated methods. We have designed our techniques to fit the kinds of dataflow programs that real users tend to write, while at the same time leaving our overall approach fairly general so as to be readily adaptable to other contexts.

1.3 Outline

The remainder of this paper is structured as follows. We first supply some background on dataflow programs and illustrative example data, and discuss strawman approaches to generating example data automatically, in Section 2. We then formalize the example generation problem in Section 3, and present our algorithmic solution in Section 4. We evaluate our solution empirically in Section 5. We summarize our findings in Section 6, and then discuss related work in Section 7. Lastly, in Section 8 we mention some opportunities for future work.

2. PROBLEM OVERVIEW

In this section we describe the problem of generating example data for dataflow programs, and give some insights into the challenges inherent in the problem by describing two strawman approaches and their drawbacks.

2.1 Input and Output

The input to an example generation algorithm is a dataflow program P over database D. In this paper we take P to be a tree of n operators, where each operator consumes one or more input tables and produces an output table. A table is a multiset of records. A record is a list of data values, each of which is either a scalar or a table. (The use of nested tables facilitates grouping as a distinct operation from aggregation, as shown in Figure 2.)

Our approach is quite general, but in this paper we focus on the core operators of the Pig Latin language [13]:

- LOAD**:** Read the content of a *base table*, i.e., a table stored in D.
- FILTER**:** Eliminate unwanted records, according to either a built-in logic predicate, or a user-supplied Boolean function that accepts or rejects each record.
- GROUP**:** Partition input records into disjoint groups, and form one output record per group. Each output record consists of a group identifier followed by a nested table containing the set of input records belonging to the group. (See Figure 2 for an example.)

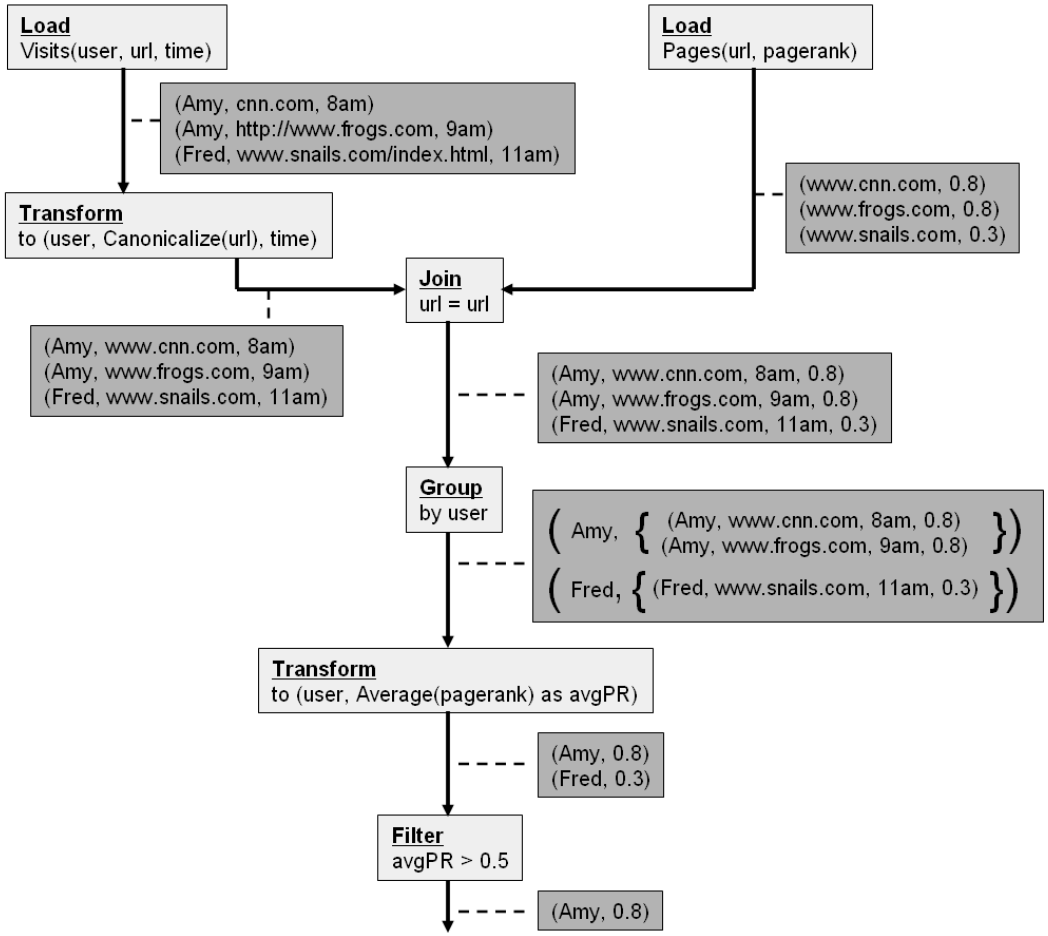

Figure 2: Dataflow program with automatically-generated example data.

- TRANSFORM**:** An arbitrary transformation function $f(\cdot)$ applied to each input record; $f(\cdot)$ may produce zero or more output records on each invocation.[2] This is a very general and flexible operator, designed to fill the role played by general MAP and REDUCE functions in the Map-Reduce architecture [7]. The TRANSFORM operator can be used for projection, built-in arithmetic transformations such as incrementing a numeric value, general user-defined transformations (e.g., the user-supplied `Canonicalize(·)` function in Figure 2), or aggregation. Aggregation is accomplished by first invoking a GROUP operation, which outputs one record per distinct group containing a nested table of the grouped-together records, and then invoking a TRANSFORM operation that aggregates the contents of each group record's nested table. Figure 2 shows an example of aggregation: the `Average(·)` function is applied to the set of records associated with each distinct user.
- JOIN**:** The relational equijoin operation: identify related pairs of records from two input tables according to an equality predicate, and combine each such pair into a single output record.

[2]In Pig Latin, TRANSFORM is invoked via the "foreach" syntax.

- UNION**:** Place each record from each of two input tables into a single output table.

If Operator A's output table is one of Operator B's input tables, we say that Operator A is an *upstream* neighbor to Operator B. Conversely, Operator B is the *downstream* neighbor of Operator A. Operators that have no upstream neighbors are *leaf operators*, and must be of type LOAD. The final operator, which has no downstream neighbor, is the *root operator*.

The output of an example generation algorithm is a set of *example tables* $\{T_1, T_2, \ldots, T_n\}$, one corresponding to the output of each operator in P, as illustrated in Figure 2. The set of example tables is required to be *consistent*, meaning that the example output table of each operator is exactly the table produced by executing the operator over its example input table(s).

Another requirement might be conformity to registered integrity constraints, such as functional dependencies. Our present work does not deal with integrity constraints, because our application scenario (Pig) does not include mechanisms for registering or enforcing constraints.

2.2 Objectives

As alluded to in Section 1, there are three primary objectives in selecting example data (formalized in Section 3.2):

- **Realism.** The example tables that correspond to original database tables (`Visits` and `Pages` in Figure 2) should be subsets of the actual database tables, if possible. If not, then to the extent possible the individual attribute values should be ones found in the actual database.
- **Completeness.** The example tables should collectively illustrate the key semantics of each operator. For example, the example tables before and after the GROUP operator in Figure 2 serve to illustrate the semantics of grouping by user, namely that input records about the same user are placed into a single output record. As another example, the example tables before and after the FILTER operator illustrate the semantics of filtering by average pagerank, namely that input records with low average pagerank are not propagated to the output. Completeness is defined on a per-operator basis (see Section 3).
- **Conciseness.** The example tables should be as small as possible, to effectively present in a user interface, and to minimize the amount of data the user has to examine.

2.3 Strawman Approaches

2.3.1 Downstream Propagation

A tempting approach is simply to take a sample of each base table, push the sample data through the operator tree, and record the intermediate table produced by each operator. Unfortunately, as mentioned in Section 1, in the presence of highly selective operators this approach may not achieve good completeness. Using larger samples may overcome this problem, but at the risk of hurting conciseness. Besides, taking very large samples would cause the example generation process to take too long. We seek algorithms that generate example data in real time as the user formulates and refines her program.

Throughout this paper we assume that any database samples used in example generation are of small size. We also assume the sampling process cannot be biased to meet the needs of a given program: As discussed in Section 1 our work targets ad-hoc programs, for which the indexes necessary for efficient biased sampling may not exist a priori.

2.3.2 Upstream Propagation

A second natural approach is to work backward: given a desired output characteristic (e.g., nonempty output table), select input records from the database samples, or else synthesize input records, that cause this characteristic to be met. This idea can be applied recursively, starting from the root operator and moving upstream toward the leaves.

Upstream propagation relies on *invertiblity* of operators, i.e., the ability to produce input table(s) that, when pushed through the operator, produce a given output table. Operator invertibility is impeded by the presence of *user-defined functions* (UDFs), which are common in operators such as TRANSFORM and FILTER. Since UDFs contain arbitrary code, it is not always possible to construct input record(s) that will lead to a given output. The only way to generate example data for a noninvertible operator is to push data through the operator in the downstream direction.

Clearly, neither upstream nor downstream propagation alone is sufficient to generate good example data, in general. However, careful interleaving of downstream and upstream propagation, combined with pruning of redundant examples, can lead to better results, and in fact this approach forms the basis for our algorithm described in Section 4.

3. PROBLEM FORMALIZATION

We now formalize the example generation problem. Of the three objectives outlined in Section 2.2, completeness presents the greatest challenge for formalization, for two reasons: (1) individuals may disagree as to what constitutes the "key semantics" of a particular operator; (2) an overly elaborate formal definition can be hard to work with algorithmically. In view of these issues we have settled on the following approach: (1) rather than attempting to formulate a universal completeness definition, we supply a general framework into which one may insert the desired completeness semantics on a per-operator basis; (2) our framework for defining completeness is based on a simple yet flexible model of record equivalence classes, described next.

3.1 Equivalence Class Model

For the purpose of formalizing our problem, we introduce the following model of how operator semantics are to be illustrated via example data.

We are given a set of operators, such as the ones listed in Section 2.1. For each operator O, we specify a set of equivalence classes $\mathcal{E}_O = \{E_1, E_2, \ldots, E_m\}$ over records, where each equivalence class is meant to illustrate one aspect of the operator semantics. Each input or output record for operator O is either not a member of any equivalence class in $\mathcal{E}_O$, or is a member of exactly one equivalence class $E_i \in \mathcal{E}_O$. Completeness for operator O is defined as having example input and output tables that collectively contain at least one member of each equivalence class in $\mathcal{E}_O$.

For example, one way to define completeness for a FILTER operator is to assign all input records that pass the filter to class E_1 and all records that do not pass the filter to E_2. Hence, the example input table for a filter is complete iff it contains at least one record that passes the filter and one that does not. If the filter is multifaceted, i.e., its predicate combines multiple primitive expressions via logical connectives, one may choose to define more than two equivalence classes—perhaps up to one equivalence class per unique combination of truth values for the primitive expressions.

3.2 Quantitative Objectives

Given the above equivalence class model, we can state our objectives from Section 2.2 quantitatively:

- **Realism** $\in [0, 1]$**:** the fraction of example records that are *real*. A record produced by a LOAD operator is real iff it appears in the corresponding table in D; a record produced by any other operator is real iff every record in its *lineage* is real.[3] (The lineage of a record is the set of records from which it has been derived in P.)
- **Completeness** $\in [0, 1]$**:** the average of per-operator completeness values, where per-operator completeness is defined as the fraction of operator equivalence classes for which at least one example record exists.
- **Conciseness** $\in [0, 1]$**:** the average of per-operator conciseness values, where per-operator conciseness is defined

[3]A possible refinement is to give "partial credit" to records containing a mixture of synthetic and real values, but for simplicity we define realism at the granularity of full records.

as the ratio of the number of operator equivalence classes to the total number of example records for that operator (with a ceiling at 1). (The use of the number of equivalence classes in the numerator avoids penalizing operators that require a large number of examples to illustrate their semantics.)

3.3 Discussion

The nature of this problem is such that no solution can guarantee a satisfactory outcome on all three objectives for arbitrary programs and data sets. In other words, even a theoretical optimal algorithm cannot, in general, achieve a score of 1 on each of realism, conciseness and completeness simultaneously.

For example, consider a program that performs grouping, then filters out groups that have cardinality below 1000, and finally takes the average of each group that passes the filter (i.e., large groups). Clearly, if we want to illustrate the final transformation (taking the average), we need to show at least 1000 input records, which violates conciseness. Hence it is impossible to simultaneously achieve completeness and conciseness in this case. In general there can be tensions between each pair of objectives, e.g., there may be no way to achieve completeness without synthesizing data and reducing realism.

That said, it is still worth pursuing this problem, because based on our experience it is possible to do quite well in many scenarios that arise in practice. Besides, one can often use special user-interface techniques in scenarios where the most concise example is not concise enough: If the 1000 constituent records of a group in the above example are not semantically different, they can be shown by an ellipsis that is expanded on demand.

4. OUR ALGORITHM

In this section we describe our algorithm for generating example data tables. Given the complexity of the problem, to allow an efficient solution we have taken a *best-effort* approach. Our algorithm works well in practice (see Section 5), but unfortunately we cannot offer a formal performance guarantee. In fact, due to the nature of the problem, even a theoretical optimal algorithm cannot always perform perfectly on all three objectives, as explained in Section 3.3.

Our algorithm is generic and can be instantiated for specific query processing systems depending on the set of operators supported, and the corresponding equivalence class definitions (Section 3.1). In this section we give the generic algorithm (along with a running example for concreteness). In Appendix A we give the specific instance of our generic algorithm that we have developed for Pig [13].

Before proceeding, we review the concept of *data lineage* [16], of which our algorithm makes heavy use. A record's *lineage* is the set of upstream records that influence the production of a given downstream record. The set of base, intermediate, and final records flowing through a dataflow program may be divided into a one or more disjoint *lineage groups* as follows. Let $\mathcal{G}$ be an undirected graph with vertices corresponding to records, and lineage relationships among records as edges. Each connected component in $\mathcal{G}$ forms a lineage group. For example, the set of records that get placed into a single group record, along with the group record itself, are part of a single lineage group.

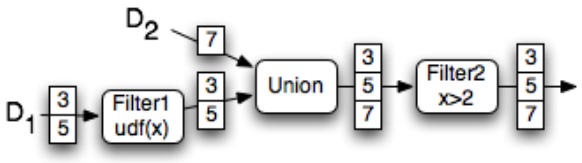

Figure 3: Downstream pass example.

4.1 Algorithm Overview

Our algorithm performs a series of four passes over a given dataflow program P, with the fourth pass yielding the final output (example data tables). Each of the four passes is described in detail in the following sections. The basic outline of the algorithm is as follows:

1. A first *downstream* pass picks arbitrary data from the input data set to work with. The chosen data is propagated through the dataflow graph providing as many examples as possible in an acceptable amount of time. (Section 4.2)

2. The data from the previous pass can be thought of as a set of disjoint lineage groups. A *pruning* pass eliminates lineage groups that are *redundant*, i.e., they harm conciseness without improving completeness. (Section 4.3)

3. In spite of the passes above, there might be some incompleteness, i.e., some equivalence classes may not have been illustrated due to the presence of selective operators. Our third *upstream* pass remedies such incompleteness by synthesizing additional data that would boost completeness. This pass is only best-effort since in the presence of noninvertible operators, it may be impossible to determine what data would boost completeness. (Section 4.4)

4. Lastly, an additional pruning pass similar to Pass 2 is conducted to eliminate redundant records introduced during Pass 3. However, since we now have both real and synthetic data, we favor retention of real data over synthetic data to boost realism. (Section 4.5)

4.2 Downstream Pass

In our downstream pass, we first take a sample of size S from each input base table (i.e., each leaf operator in P).[4] We then evaluate P over the sampled input records, and record the intermediate table produced between each pair of operators as well as the final output table generated by the root operator.

EXAMPLE 1. *We use the very simple dataflow graph shown in Figure 3 as a running example to illustrate the stages of our algorithm. There are two input data sets D_1 and D_2 with a single atribute x. Data set D_1 is filtered by a user-defined function $udf(\cdot)$, then unioned with D_2. The result is filtered by the condition $x > 2$ to get the final result. The downstream pass is simple: arbitrary data is picked from*

[4]In principle any form of sampling may be used here. In practice the sampling must be done efficiently, to ensure low latency for the overall example generation algorithm to return to the user. Uniform random samples are not required. In our implementation we simply read the first 10,000 records from each data file.

D_1 and D_2 and propagated through the dataflow graph. Note that in this example, the records 3 and 5 happen to pass the filter defined by udf.

4.3 Pruning Pass

The pruning pass is given a program P and the set of input, intermediate and output data tables generated by the downstream pass (e.g., the content of Figure 3). The goal is to enhance conciseness by eliminating lineage groups that are large and redundant with respect to completeness. Our pruning algorithm examines one operator at a time, starting with the root operator and proceeding upstream toward the leaves. Without loss of generality, for simplicity of exposition we assume the operator tree consists of a linear chain of operators $O_1, O_2, \ldots, O_n$, where O_n is the final root operator and O_1 is a leaf operator. The pruning algorithm starts with O_n, then proceeds to O_{n-1}, and so on until reaching O_1.

Let O_k be the operator currently under consideration. We refer to $O_{k+1}, O_{k+2}, \ldots, O_n$ collectively as the *downstream operators*, and $O_1, O_2, \ldots, O_k$ as the *upstream operators* (note that O_k is grouped with the upstream operators). For the purpose of pruning, the set of input, intermediate and output records associated with the upstream operators $O_1, O_2, \ldots, O_k$ are arranged into $m \geq 1$ lineage groups, denoted $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$. The pruning procedure at operator O_k may eliminate one or more of these upstream lineage groups.

4.3.1 Completeness Constraints

To ensure that completeness is not diminished in the process, pruning is guided by *completeness constraints*. A completeness constraint C is a set of records $\{r_1, r_2, \ldots\}$ and an associated *minimum coverage* number $M_C \geq 1$, which specifies that eliminating more than M_C members of C causes a reduction in completeness. The number of members of C that are retained by the pruning algorithm is called C's *coverage level*. The coverage level of each completeness constraint is to be kept equal to or above the minimum coverage level M_C.

There are two kinds of completeness constraints: *Upstream completeness constraints* encode the completeness requirements of the upstream operators. *Downstream completeness constraints* encode completeness requirements of the downstream operators.

We first consider upstream completeness constraints. Recall from Section 3.1 that associated with each operator O is a grouping of records into equivalence classes $\mathcal{E}_O = \{E_1, E_2, \ldots\}$. By definition, preserving any single member of an equivalence class is sufficient to maintain the same level of completeness. As such, each upstream equivalence class E_i forms an upstream completeness constraint C_i with minimum coverage $M_{C_i} = 1$.

Downstream completeness constraints range over operator O_k's output records, but they encode the completeness semantics on behalf of all downstream operators $O_{k+1}, O_{k+2}, \ldots, O_n$. They are propagated upstream as pruning moves upstream. In particular, after pruning of operator O_k is complete, the downstream completeness constraints, which are over O_k's output records, are transformed into constraints over O_k's input records, i.e., O_{k-1}'s output records, for use in the next round of pruning. This constraint transformation process is governed by operator-specific logic

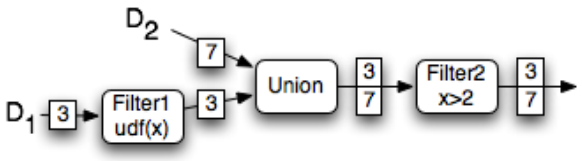

Figure 4: First pruning pass example.

associated with O_k; Appendix A.3 gives completeness constraint propagation logic we use in our Pig-based implementation.

4.3.2 Single-Operator Pruning Algorithm

Given the upstream lineage groups $\mathcal{L}$ and completeness constraints $\mathcal{C}$ associated with operator O_k, pruning can be formulated as a simple optimization problem: Select a subset of lineage groups $\hat{\mathcal{L}} \subseteq \mathcal{L}$ to retain, such that each completeness constraint $C \subseteq \mathcal{C}$ is covered by at least M_C records in $\hat{\mathcal{L}}$, and the following function is minimized:

$$\sum_{L \in \hat{\mathcal{L}}} w(L)$$

where $w(L) = \sum_{r \in L} w(r)$ and $w(r)$ is a per-record weight. In the first pruning pass, we use uniform weights across all records; we shall use nonuniform weights in our second pruning pass (Section 4.5).

If all $M_C = 1$, this optimization problem is isomorphic to the classical (weighted) set-cover problem, which is known to be NP-hard to approximate to a factor better than $O(\log n)$ [6, 10], where in our case n is the number of completeness constraints ($n = |\mathcal{C}|$). The following greedy algorithm achieves the optimal approximation factor $O(\log |\mathcal{C}|)$:

Initialize $\hat{\mathcal{L}} = \{\}$. Repeatedly add to $\hat{\mathcal{L}}$ the lineage group $L \in \mathcal{L} \setminus \hat{\mathcal{L}}$ with the highest value of $c(L)/w(L)$, where $w(L)$ is as defined above, and $c(L)$ is the number of completeness constraints covered by L but not covered by any group already in $\hat{\mathcal{L}}$:

$$c(L) = |\{C \in \mathcal{C} : (C \cap L \neq \{\}) \wedge (\forall L' \in \hat{\mathcal{L}}, C \cap L' = \{\})\}|$$

Stop when all completeness constraints have been covered, i.e., $\forall C \in \mathcal{C}, \exists L \in \hat{\mathcal{L}} : C \cap L \neq \{\}$.

In the presence of completeness constraints having minimum coverage $M_C > 1$, the set-cover algorithm can be generalized in the obvious way: Let $c(L)$ be the number of completeness constraints C whose coverage level was below M_C, and increased as a result of adding L:

$$c(L) = |\{C \in \mathcal{C} : (C \cap L \neq \{\}) \wedge (|\{r \in C : (\exists L' \in \hat{\mathcal{L}} : r \in L')\}| < M_C)\}|$$

Stop when all completeness constrains have reached or exceeded minimum coverage, i.e., $\forall C \in \mathcal{C}, |\{r \in C : (\exists L \in \hat{\mathcal{L}} : r \in L)\}| \geq M_C$.

EXAMPLE 2. *For the purpose of our running example, say our pruning algorithm operates according to the following definition of equivalence classes for the filter operator: all records that pass the filter form equivalence class E_1 and those that don't pass the filter are in class E_2. Among the example data at the end of the dowstream pass (Figure 3), records 3 and 5 belong to the same equivalence classes (they both pass both the filters). Thus, one of them (say 5) is pruned away to yield the example data as shown in Figure 4.*

4.4 Upstream Pass

The upstream pass begins at the root operator, and proceeds recursively toward the leaves. At each operator, it identifies cases of incompleteness (i.e., empty equivalence classes). When incompleteness is detected, the algorithm attempts to manufacture a *constraint record*, which describes the set of possible records that would remedy the incompleteness. Constraint records are inserted into the operator's input, and propagate upstream as the algorithm moves upstream. When the algorithm reaches a leaf operator, it converts constraint records into concrete records, and attempts to conform as closely as possible to real data available at the leaves.

The details of the upstream pass are given below, starting with a definition of constraint records.

4.4.1 Constraint Records

A *constraint record* is a concise representation of a set of possible records [14]. The set of possibilities may be defined explicitly (i.e., a list of alternatives), or implicitly (i.e., a system of constraints over the data contained in the record); in the latter case the set may be infinite.

In our current work we adopt the following simple language for constraint records: a list of fields, where each field contains one of (a) a concrete value for that field, or (b) a "don't care" marker denoting that any value is acceptable. This constraint language may appear overly simple, but it has proven adequate in our experience in terms of allowing us to generate good example data sets (see Section 5), and its simplicity makes it easy to deal with in our code. Our techniques are general in that they can be extended to incorporate richer constraint languages; we leave this as a topic of future work.

4.4.2 Algorithm

As mentioned earlier, our example generation algorithm is heuristical, i.e. it does not guarantee to find an optimal solution. The upstream pass has two aspects that may lead to suboptimal results: (a) operator-at-a-time processing with no lookahead, and (b) limited constraint language. In these respects we have favored simplicity and efficiency over exhaustiveness. The upstream algorithm is as follows:

1. Let O be the operator currently under consideration. At the outset, O is set to the root operator.
2. For each constraint record R in O's output table (in the case of the root operator, this set will be empty), if possible insert constraint records into O's input table(s) such that their fulfillment would lead to fulfillment of R.
3. For each empty equivalence class E associated with O, if possible introduce constraint records into O's input table(s) such that fulfillment of the constraint records would cause E to become nonempty. (This step relies on operator invertibility; if this step fails due to noninvertibility, simply skip it and move on.)
4. Repeat Steps 2 and 3 recursively, with O moving toward the leaves of the operator tree.
5. When a leaf operator O is reached, after performing the processing described in Steps 2 and 3, convert each constraint record in O's input table into an actual record, in such a way as to conform to real data values found in the base table as much as possible. Under our simple constant/"don't care" constraint language specified in Section 4.4.1, we simply replace each "don't care" field of a constraint record with a value sampled from the corresponding field in the base table.
6. Discard all intermediate tables and the output table, as well as all constraint records that have not been successfully converted into concrete records.
7. Re-evaluate P over the newly-augmented input tables, and record the new intermediate tables and new output table.

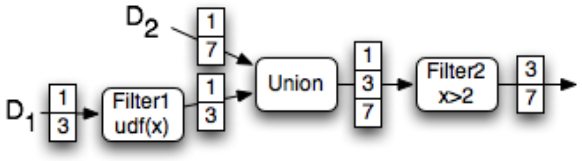

Figure 5: Upstream pass example.

Steps 2 and 3 require per-operator rules for crafting constraint records to insert into an operator's input table(s). Two types of upstream propagation rules are required: (1) converting output constraint records received from the downstream neighbor into input constraint records to give to the upstream neighbor(s), and (2) creating input constraint records to populate empty equivalence classes. The rules are specific to the particular equivalence class definitions being applied, and tend to be of a best-effort nature. In Appendix A we present a set of concrete equivalence class definitions and corresponding upstream propagation rules, to give an example of how our algorithm can be instantiated.

EXAMPLE 3. *In our running example, the example data at the end of the first pruning pass (Figure 4) did not cover all equivalence classes (there are no examples of records not passing the filters). Hence the upstream phase attempts to cover all equivalence classes by synthesizing additional data. Starting at the root operator (Filter2), we find that equivalence class E_2 (records that do not pass the filter) is empty. Hence the upstream phase synthesizes a new record (say 1) that will not pass the filter. This synthetic record is propagated upstream. To give upstream propagation maximum choice, 1 is propagated to both branches of the union operator. Finally, we find that 1 also happens to pass the udf filter. The upstream pass attempts to find an example tuple that will not pass the udf filter, but is unable to do so because of the noninvertibility of udf$(\cdot)$. The final data at the end of the upstream pass is as shown in Figure 5.*

4.5 Second Pruning Pass

The final pass reapplies the pruning algorithm of Section 4.3 to eliminate any redundant lineage groups introduced in the upstream pass. Since the data now contains a mixture of real and synthetic records, we assign weights to records with the goal of biasing pruning toward eliminating synthetic records (and hence retaining real ones). In particular, we assign $w(r) = 1$ if r is a real record and $w(r) = \alpha$ if r is a synthetic record. ($\alpha \geq 1$ is a parameter that governs

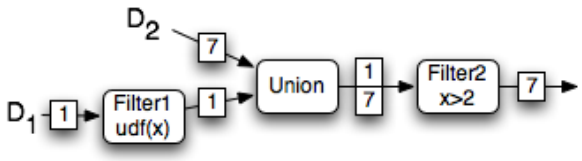

Figure 6: Second pruning pass example.

the relative preference for real data over synthetic data; in our experiments we use $\alpha = 2$.)

EXAMPLE 4. *In our running example, the example data at the end of the upstream pass (Figure 5) contains some redundant records. The pruning procedure yields the final example data as shown in Figure 6. Note that even in this extremely simple example, our algorithm does well on conciseness but lacks perfect realism (one of the records is synthetic), and perfect completeness (there is no example to illustrate when udf would not pass). However, as discussed in Section 3.3, doing perfectly well on all the three metrics is generally impossible for any algorithm.*

5. EXPERIMENTS

In this section we evaluate our approach quantitatively, on real programs and data used at Yahoo! The programs are written in Pig Latin. For confidentiality reasons we cannot give details of the programs or data. Instead we show the abstract dataflow structure of each program, and disclose features of these programs that get in the way of automatic example generation. We then present quantitative results over these workloads.

5.1 Dataflow Programs

Our real-life dataflow programs have the following abstract structure:

Program 1 (web search query frequency over time):

1. LOAD table A
2. FILTER using a UDF (**noninvertible**)
3. TRANSFORM using a UDF (**noninvertible**)
4. GROUP
5. TRANSFORM using a built-in aggregation function
6. FILTER via arithmetic comparison
7. GROUP
8. TRANSFORM using a built-in aggregation function

Program 2 (news web site usage statistics):

1. LOAD table B
2. FILTER via arithmetic comparison (**highly selective**)
3. GROUP
4. TRANSFORM using a built-in aggregation function

Program 3 (web search result viewing statistics):

1. LOAD table C
2. FILTER using a compound arithmetic comparison
3. GROUP
4. TRANSFORM using a built-in aggregation function

Program 4 (web search result viewing statistics):

1. LOAD table C
2. FILTER using a compound arithmetic comparison
3. GROUP
4. TRANSFORM using a built-in aggregation function
5. GROUP
6. TRANSFORM using a built-in aggregation function

Program 5 (web search result viewing statistics):

1. LOAD table C
2. FILTER using a compound arithmetic comparison
3. GROUP
4. TRANSFORM using a built-in aggregation function
5. GROUP (**forms very large groups**)
6. TRANSFORM using two built-in aggregation functions

Program 6 (web advertising query frequency):

1. LOAD table D
2. LOAD table E
3. JOIN D and E w/equality predicate (**highly selective**)

Program 7 (web advertising human judgments):

1. LOAD table F
2. FILTER F using a compound logical expression
3. LOAD table G
4. JOIN F and G w/equality predicate (**highly selective**)
5. TRANSFORM using four string transformation UDFs (**noninvertible**)

Program 8 (web advertising activity):

1. LOAD table H
2. FILTER H using a compound logical expression
3. LOAD table I
4. JOIN H and I w/equality predicate (**highly selective**)
5. TRANSFORM using four string transformation UDFs (**noninvertible**)

The above programs contain instances of each of the main challenges we have identified in this paper: selective operators, which poses a problem for downstream propagation, noninvertible operators, which poses a problem for upstream propagation, and group operators whose groups are all large, which poses a problem if pruning is applied only at the root level (see Section 5.2).

5.2 Example Generation Algorithms

We tried four algorithms (two baseline algorithms, and two new ones):

- **Downstream:** propagate a database sample downstream through the operator tree (Section 2.3.1).
- **Upstream:** propagate constraints upstream through the operator tree (Section 2.3.2).
- **Root-level pruning:** our algorithm (Section 4), with pruning performed at the root operator only, rather than iteratively pruning each operator via an upstream tree traversal as in our full algorithm.
- **Our algorithm:** our algorithm (Section 4), with full pruning.

The rationale for including the root-level pruning variant of our algorithm is to determine whether full pruning is worth the additional cost.

For both variants of our algorithm, we found that taking the logarithm of the lineage group weights yielded somewhat better results, compared with using raw weights. By taking the logarithm, we de-emphasize the weights, causing our algorithms to favor a smaller number of groups over groups with a small number of records.

We also found that our algorithms are not very sensitive to the parameter α (Section 4.3.2); we use $\alpha = 2$ in all of our experiments. For all algorithms that involve a downstream propagation step, we use a sample size of 10,000 base table records.

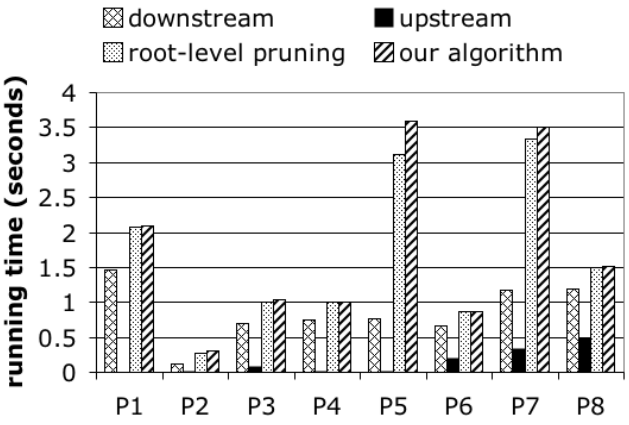

Figure 7: Algorithm running times.

5.3 Running Time

We implemented the four algorithms in Java, and did not spend any time tuning them for fast execution. Figure 7 shows the running time of each algorithm on each program, when run on a modern commodity workstation. The upstream algorithm is by far the fastest, because it does not perform an initial downstream propagation step with 10,000 records as done by the other three algorithms. The downstream propagation step takes up to 1.5 seconds on these programs, as indicated by the performance of the downstream algorithm in Figure 7. As expected, our algorithms add additional overhead, bringing the total execution time to at most 3.5 seconds on the programs we studied. A response time of 3.5 seconds to produce example data is acceptable for an interactive end-user application, although ideally with a bit of performance tuning we can reduce this figure.

5.4 Quality of Generated Example Data

Figure 8 shows the result of running all four algorithms on all eight programs, under our three metrics given in Section 3.2.

The downstream algorithm performs well on realism, of course, since it does not synthesize data. However it is unable to achieve simultaneously good conciseness and completeness. For example, on Program 1, it achieves good completeness by luck, due to the sheer number of records pushed through the program, but of course conciseness is extremely poor (approaching zero). Program 2 represents the opposite extreme, in which a highly selective operator early in the program keeps conciseness high but leads to poor completeness.

The upstream algorithm fares better on completeness for Program 2 by introducing synthetic data, thereby decreasing realism. On Program 1, the upstream algorithm performs poorly on both completeness and realism relative to the downstream algorithm, due to the presence of a noninvertible operator blocking the upstream propagation of constraints.

The behavior of the downstream and upstream algorithms on Programs 3–5 is similar to their behavior on Program 1. On Programs 6–8, which involve selective joins, the downstream algorithm does not always achieve good completeness because of the selectivity of the join. The upstream algorithm also struggles with completeness on these programs, although our simplistic constraint language is to blame. This situation could be remedied by moving to a more sophisticated constraint language able to express equality constraints among synthetic data values. However, even with such limited expressibility in the constraint language, our algorithms are able to achieve perfect completeness scores on these programs by matching real data values with synthetic ones, and this simplicity ensures that the algorithm is sufficiently fast (see Section 5.3).

The root-level pruning variant of our algorithm does not prune as aggressively as our full algorithm, and consequently achieves lower conciseness in a few cases. Given that the full algorithm incurs only modestly higher running time than the root-level variant (Section 5.3), it seems worthwhile.

Comparing our full algorithm against the downstream and upstream baselines, we see substantially better results. Completeness is the most important metric, i.e., without good completeness the other metrics are irrelevant. As such, completeness is treated as a hard constraint in our approach, whereas realism and conciseness are soft constraints. On all eight programs the level of completeness achieved by our algorithm matches or exceeds the better of downstream and upstream. The conciseness and realism levels tend fall in between the levels for downstream and upstream, matching the better of the two baselines in about half the cases.

In absolute terms, our algorithm yields very good results: Completeness is at or near 100% in seven out of eight cases, and in the remaining case it is over 50%. In half the cases 100% realism is achieved, and three of the four remaining cases have a roughly even mixture of real and synthetic data. In all cases conciseness is close to or above 50%—a 50% conciseness score means that example tables are only twice as large as necessary to illustrate the program semantics. (Recall from Section 3.3 that even a theoretical optimal algorithm cannot always achieve full conciseness along with full completeness for a given program.)

6. SUMMARY

While formulating a dataflow program, it is helpful to trace a handful of example data records as they flow through the operators, to verify that each operator is behaving as intended. This kind of example-driven development requires the ability to concoct a small set of realistic records, that collectively illustrate the operator behaviors.

In this paper we formalized this problem and presented an algorithm that generates example data records for this purpose. Using experiments over real programs and data, we demonstrated the effectiveness of our algorithm, as well as the ineffectiveness of two baseline approaches.

Our example generation algorithm is fully implemented and released as an open-source contribution to the Apache Pig dataflow system at `http://hadoop.apache.org/pig`. It can be invoked via the ILLUSTRATE command from the Pig shell.

7. RELATED WORK

The idea of generating example data to illustrate data processing semantics appeared some time ago in [12]. The work in [12] focused on illustrating the semantics of an entire query block as a unit, whereas our work focuses on illustrating each intermediate result of a dataflow program. Consequently the techniques are very different. In particular our approach performs a series of forward and backward passes over the dataflow graph, and attempts to achieve suitable

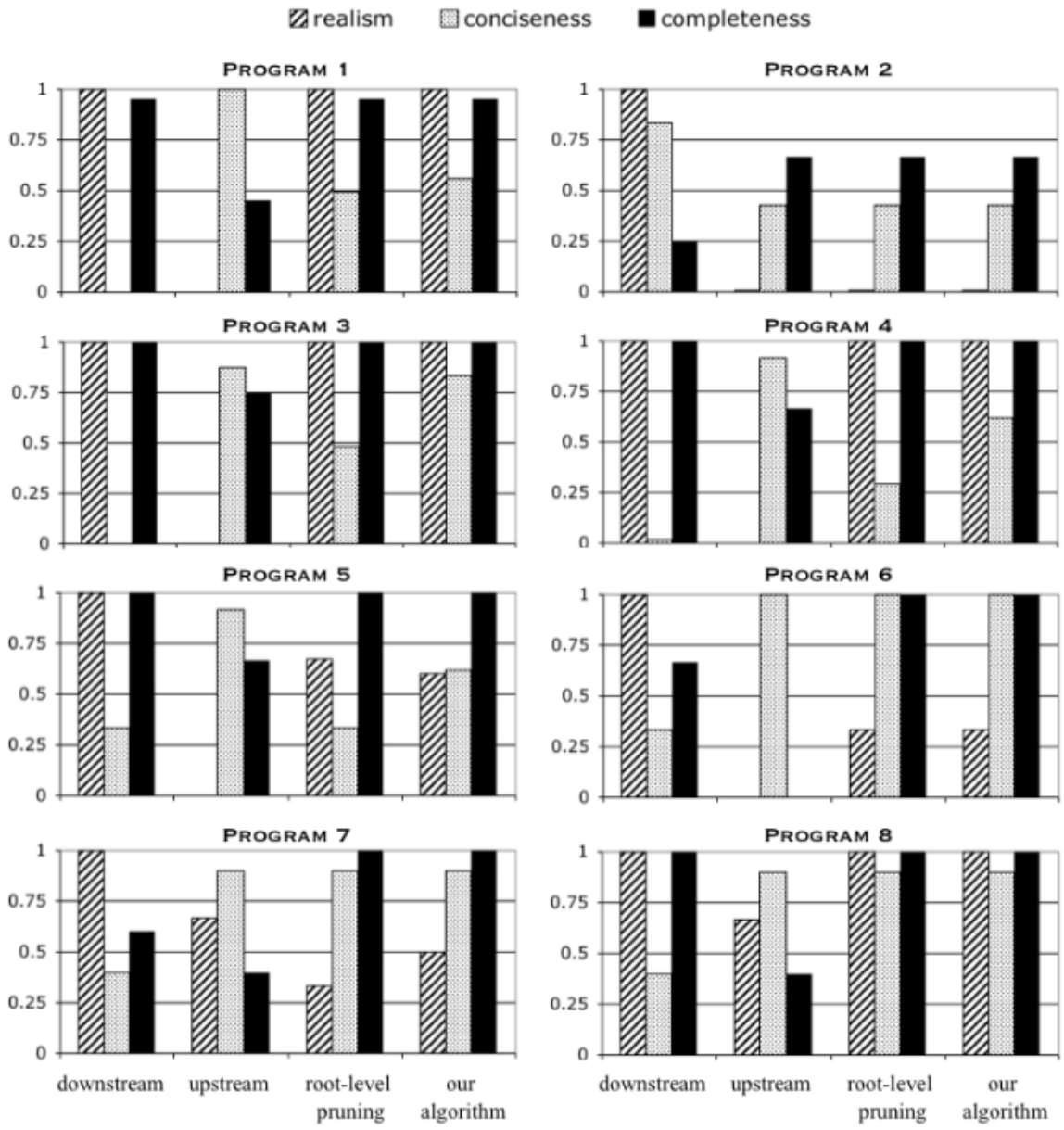

Figure 8: Example generator output quality.

intermediate example tables at each level simultaneously, working within the constraint of mutual consistency across the intermediate tables. In contrast, [12] focuses only on the last level (the final query output) and on making sure all the functional dependencies are illustrated there.

Recent work on reverse query processing [3] can potentially be applied to the "backward pass" portion of our approach. Indeed, [3] does mention some applications in verification and post-hoc debugging, which are related to our scenario of generating illustrative example data but at the same time impose rather different requirements. In particular, verification is not driven by conformity to a pre-existing input database, and post-hoc debugging does not involve the kind of synthesis and pruning of data that our context necessitates.

There is recent work on synthesizing test databases in order to uncover DBMS implementation bugs. The approach in [4], which tailors the test database to a particular query, bears some resemblance to the one we propose in this paper, namely its use of symbolic query processing over constraint records, in the spirit of constraint databases [14]. However the factors that drive the symbolic processing are quite different from our case: In [4] the aim is to ensure that intermediate record sets conform to a particular distribution and cardinality, whereas our processing aims to produce records that illustrate the key semantic properties of each operator, while conforming as closely as possible to records found in a pre-existing database. Also, the notion of conciseness is different: in automated test generation the goal is to minimize test execution time, whereas in our context the goal is to minimize the size of intermediate result sets so that they can be effectively presented in a user interface.

In the formal verification and software engineering communities there has been extensive work on automatic gen-

eration of test cases for general software and hardware systems, much of it well before the aforementioned work on automated DBMS testing; see, e.g., [8, 11]. At a high level the goals are similar to ours, namely to generate a concise suite of input data instances that collectively exercise all aspects of a given program or circuit. Therefore, not surprisingly, our methods bear some resemblance to the methods used in software and hardware testing, although there are fundamental differences as we point out below.

Many hardware verification techniques perform forward and reverse propagation of constraints, as in our approach. However, whereas hardware verification deals with abstract $\{0, 1\}$ data values, our approach focuses on conforming as closely as possible to an underlying input database to maximize "realism," which is not a factor in hardware verification. Moreover, in our work a principal objective is to minimize intermediate data sizes in the presence of set-valued operators, which do not arise in hardware circuits.

As with hardware verification, software testing also lacks the notions of conformity to pre-existing input data and minimizing intermediate data sizes, both of which are central to our work. That said, it should be possible to adopt some of the sophisticated logical and arithmetic reasoning algorithms from software model-checking to our context, to improve upon our simple "don't care" constraint language.

8. FUTURE WORK

Directions for future work include:

- **Coherence across program modifications.** When a program is being constructed iteratively, it is desirable to maintain some coherence of the example data across iterations. In other words, if a certain example record is shown to the user for iteration i of program construction, all else being equal it would be good to keep showing the same example record throughout all subsequent iterations $i + 1, i + 2, \ldots$.
- **Richer constraint language for constraint records.** Adopting a richer constraint language for constraint records (e.g., arithmetic predicates like "$X < 5$") may lead to better results. Techniques from constraint databases [14], symbolic query processing [4] and reverse query processing [3], are likely to be of use here.

9. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Aug. 2003.

[2] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computing Systems*, 21(1):36–86, Feb. 2003.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proc. ICDE*, 2007.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Ozsu. QAGen: Generating query-aware test databases. In *Proc. ACM SIGMOD*, 2007.

[5] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD*, 1999.

[6] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4, 1979.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[8] P. Goel and B. C. Rosales. An automatic test generation system for VLSI logic structures. In *Proc. Conference on Design Automation*, 1981.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. European Conference on Computer Systems (EuroSys)*, 2007.

[10] D. S. Johnson. Approximation algorithms for combinatorial problems. *Computer and System Sciences*, 9, 1974.

[11] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8), Aug. 1990.

[12] H. Mannila and K.-J. Raiha. Test data for relational queries. In *Proc. PODS*, 1986.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[14] P. Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[15] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *Proc. VLDB*, 1993.

[16] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. ICDE*, 1997.

APPENDIX

A. AN EXAMPLE GENERATOR FOR PIG

Given our generic algorithm described in Section 4, we now present a concrete instantiation that we have developed for Pig Latin [13]. It is by no means the only possible, or even reasonable, instantiation for Pig Latin, because as we mentioned in Section 3 there are many reasonable ways to define completeness, but it is one we have designed to meet the needs of our user community. Recall from Section 3 that completeness is defined via per-operator equivalence class definitions, presented next.

A.1 Equivalence Class Definitions

- LOAD**:** Every record in the base table being loaded is assigned to a single equivalence class E_1.
- FILTER**:** Every input record that passes the filter is assigned to a class E_1; all others are assigned to E_2. (The intention is to show at least one record that passes the filter, and one that does not pass.)
- GROUP**:** Every record that can be produced by the operator, and whose nested table contains at least two records, is assigned to a class E_1; other records are not assigned to any equivalence class. (The purpose of E_1 is to illustrate a case where multiple input records are combined into a single output record.)
- TRANSFORM**:** Every input record is assigned to a single equivalence class E_1. (The intention is to illustrate at least one application of the transformation.)

- JOIN: Every output record is assigned to a single class E_1. (The intention is to illustrate a case of two input records being joined.)
- UNION: Every record from the first input table is assigned to E_1; every record from the second input table is assigned to E_2. (The aim is to show at least one record from each input table being placed into the unioned output.)

A.2 Constraint Record Propagation Rules

Recall from Section 4.4.2 that the upstream propagation pass requires per-operator rules that (1) create input constraint records to populate empty equivalence classes, and (2) convert output constraint records received from the downstream neighbor into input constraint records to give to the upstream neighbor(s). We sketch a set of rules below, which attempt to achieve completeness under the equivalence class definitions of Appendix A.1 (for brevity we omit many straightforward details).

- LOAD: Replace the "don't care" fields of each output constraint record by real data values sampled from the base table.
- FILTER: For each output constraint record received from the downstream neighbor, create a copy and check whether it passes the filter. If it does pass, then give the copy to the upstream neighbor. If it does not pass, then attempt to modify it so that it does pass by converting some of the "don't care" fields into concrete data values using heuristical satisfiability logic, and then give it to the upstream neighbor.

  After all the output constraint records have been processed, if there are no (concrete or suggestion) input records that pass the filter, attempt to create a new constraint record that does pass the filter and give it to the upstream neighbor, using heuristical satisfiability logic. Similarly, if there are no input records that fail to pass the filter, attempt to create a new constraint record that does fail to pass and give it to the upstream neighbor.
- GROUP: For each concrete or suggestion output record, which corresponds to one group and is of the form $r_G = (group\_id, \{r_1, r_2, \ldots, r_m\})$, create input constraint records as needed to ensure that the group contains at least two records (i.e., $m \geq 2$). (In this way we ensure that each group r_G has adequate membership to be placed into equivalence class E_1 (see Appendix A.1), thereby giving the subsequent pruning phase many viable lineage groups to choose from.)
- TRANSFORM: To the extent possible, apply the inverse transformation to convert output constraint records into input constraint records. For example, if the transformation is a projection, add extra fields with "don't care" markers for the projected attributes. Unfortunately, as mentioned earlier, noninvertible transformations will block the upstream propagation of constraint records.
- JOIN: Separate each output constraint record into two input constraint records that, when joined, produce the output record.
- UNION: For each output constraint record, create k copies to serve as input constraint records for the k inputs being unioned. (The idea is to give the subsequent pruning phase the option to chose which branch of the program the synthesized record ought to come from, e.g., based on which one yields better realism.) If there were no output constraint records, then create k new input constraint records with "don't care" markers in every field, one for each input.

A.3 Maintaining Downstream Completeness During Pruning

Recall that in pruning (Section 4.3.1), downstream completeness constraints are used to avoid pruning data whose removal results in reduced downstream completeness, i.e., reduced coverage of downstream equivalence classes. Given the equivalence class definitions we adopt for Pig (Appendix A.1), it is difficult in general to use simple coverage constraints to guarantee that completeness is upheld in all cases. Programs that contain the equivalent of an SQL HAVING clause represent one of the problematic cases. For example, consider a program that groups web page views by user, then computes the average age, and then filters groups whose average age is less than thirty. If we need to retain a group of users whose average age is below thirty to illustrate the semantics of FILTER, then we need to constrain upstream pruning such that as it prunes individual users from within the group, it keeps the average age below thirty.

Rather than adopt a complex constraint language, we use the following simple heuristical approach. First, after performing pruning at operator O_k, we perform a *downstream completeness check*, which entails computing downstream records and verifying that every previously-covered downstream equivalence class remains covered. If not, pruning is halted and the data is rolled back to the result of the prior pruning step on O_{k+1}.

Second, to reduce the likelihood of failing the completeness check, some simple completeness constraints are introduced and propagated on a best-effort basis. In practice, the case most likely to cause a failed completeness check is one in which there is a downstream GROUP operator. Recall from Appendix A.1 that to illustrate the semantics of grouping we require that at least two records be present in a group. Hence, whenever a GROUP operator is encountered at position O_k, after pruning of O_k is complete a set of completeness constraints with $M_C = 2$, one per group retained while pruning O_k, are formed and passed to the O_{k-1} pruning step, to ensure that upstream pruning retains at least two members of each group.

Completeness constraints introduced by a GROUP operator are propagated upstream in the following best-effort manner. For FILTER and TRANSFORM operators, there is a one-to-one mapping from output records to input records, making it possible to translate constraints over the output records to ones over the input records directly. For GROUP, JOIN and UNION operators there is no way to propagate the constraints exactly so we simply do not propagate them; any problems that arise will be detected by the completeness check. Lastly, for obvious reasons there is no need to propagate constraints through a LOAD operator.