COLUMBUS: Feature Selection on Data Analytics Systems

Arun Kumar Pradap Konda Christopher Ré

February 28, 2013

Abstract

There is an arms race in the data management industry to support analytics applications over large data. Recent systems from IBM, Oracle, and SAP enable one to run R scripts over data stored in a data processing system. A key step in analytics applications is *feature selection*, which is an iterative process that involves statistical algorithms and data manipulations. From conversations with analysts at enterprise settings, we learned that feature selection is often described using logical operations, e.g., adding or removing a feature, and evaluating a set of features. But this level of abstraction for feature selection is not present in current R-based data processing systems.

Building on the literature on feature selection in practice, we organize a set of common operations for feature selection and build a prototype usable in the Oracle R Enterprise environment. We study two benefits of this level of abstraction for feature selection – *performance optimizations* and *provenance tracking*. We show how two classical database-style ideas – *batching* and *materialization* – can improve the performance of feature selection operations significantly but raise non-obvious tradeoffs. We formalize our optimizations and show that, while they are NP-Hard in general, we can obtain optimal solutions efficiently in some cases that we observed in real feature selection workloads. We empirically validate that our optimizations can improve performance by over an order of magnitude on real feature selection workloads. We then show how our framework enables us to easily track the provenance of feature selection processes and validate that the runtime overhead for capturing provenance is less than 5% on the feature selection workloads that we studied. We also discuss how we can reuse captured provenance to further improve performance.

1 Introduction

Enterprise data analytics is a booming area of the data management industry. Many enterprise applications use statistical methods, and such applications are increasingly using larger amounts of data [5–7, 13]. Consequently, there has been a massive push from the data management industry to scale statistical computing languages such as R [10–12, 26, 48], which is a familiar environment for statistical analysts. Systems like Oracle R Enterprise [10], SystemML [26] and SAP's R [12] scale the low-level primitives of R (e.g., addition and multiplication of matrices) to larger data in platforms such as RDBMS or Hadoop. The R-based systems also support routine data manipulations such as relational-style selections and projections. However, enterprise analytics is an ecosystem of several tasks and algorithms, often as an iterative process – the *analytics lifecycle* [13]. Naturally, the vendors of R-based systems are beginning to add support for analytics tasks beyond just R scripts or specific algorithms [10, 26].

We focus on one major task in the analytics lifecycle – selecting which *features* (also called signals, attributes, or variables) of the dataset to use for statistical modeling – a problem known as *feature selection* [31, 32]. Feature selection helps improve modeling accuracy, and also helps an analyst understand the importance of the features in the data. The features that are selected largely retain their meaning [30] – they typically represent real-world signals, e.g., income or age. Various algorithms for feature selection have been proposed in the statistics and machine learning literature [31, 32].

However, from interacting with analysts in enterprise settings – an insurance firm (AmFam), a consulting firm (Deloitte), an Oracle customer, and an e-commerce firm – we learned that feature selection is often practised as a process that has the analyst in the loop. The analysts use feature selection algorithms, data statistics, and data manipulations – "a dialogue" that is often specific to the application domain [13]. Analysts often use domain-specific "cookbooks" that outline best practices for feature selection [3, 14, 30]. Increasingly, analysts want to bring larger data into such processes [7]. Recent R-based systems aim to tackle the need to manage larger data [10, 12, 26], but our conversations with the analysts revealed a key problem: analysts describe feature selection at the level of *logical operations* like adding or dropping features, filtering the data, evaluating features, etc. However this level of abstraction for feature selection is not present in current R-based data processing systems, resulting in two major issues:

- *Performance*: By being aware of the computational and data access properties of logical operations, the system could better optimize its performance.
- *Provenance*: The analysts described how they have to manage information about the process – its *provenance* (steps and intermediate results) manually with spreadsheets, and even emails, thus scattering information that may have to be queried for business purposes.

We take a first step towards addressing the above two issues by using data management ideas. Based on the literature on feature selection practice [3, 14, 30] and our interactions with the analysts, we organize a set of common operations for feature selection into a framework that we name Columbus . We categorize the operations in Columbus into three types based on their role in the process – *Explore*, *Evaluate*, and *Data-Transform*. Explore operations steer feature selection manually or using selection algorithms. Evaluate operations compute numeric scores over the data. Data-Transform operations represent data manipulations, most of which are relational. We prototype Columbus operations as first-class primitives usable in the Oracle R Enterprise (ORE) [10] environment, with an R front-end and an RDBMS back-end.[1] The datasets are relational tables with the features representing attributes. Analysts perform feature selection using a sequence of Columbus operations, which constitute a feature selection *program*, also referred to as a Columbus program. We then address the two issues:

- *Performance*: We study how two classical database-style performance optimization ideas – *batching*, and *materialization* – impact feature selection programs. We formalize these two optimizations in our framework and show that they can be solved efficiently in some cases but are NP-Hard in general.
- *Provenance*: We discuss how our framework lets us easily manage the provenance of the feature selection process, and discuss how we can reuse captured results to improve performance.

[1]We also replicated our prototype with Python over PostgreSQL to show the portability of our framework.

Performance Optimizations Our main technical contribution is to study the impact of *batching* and *materialization* in optimizing the performance of feature selection programs. We observe two key characteristics of feature selection programs that make them amenable to these two optimizations. First, multiple operations might *scan the same dataset independently*. For example, stepwise selection, which is a popular feature selection operation [30], performs 10-fold cross-validation on multiple sets of features (*feature sets*) by adding one feature at a time. When selecting one out of 200 features, executing such an operation naively for, say, 5 training iterations, results in $200 \times 10 \times 5 = 10,000$ scans. Using *batching*, we can reduce the number of scans to 5. We formalize batching as an optimization problem and show that we can solve the problem efficiently in a common case where we have enough system memory to batch as many operations as needed. We show that when the system memory constrains how much we can batch, our problem is NP-Hard. Second, operations in feature selection programs might *access only a few features* from the dataset. For example, an analyst might explore only a set of 50 features out of 200 in a dataset. Using a *materialized* dataset with those 50 features projected might improve performance. But materialization presents a non-obvious tradeoff – the materialized projection is beneficial only if it is used often enough to pay off the time to materialize. We thus formulate a cost-based optimization problem to make materialization decisions using the costs of materializations and scans. We then focus on a case that we found to arise in real feature selection workloads – the input consists of a collection of feature sets that form a *chain*, i.e., the feature sets form a sequence where a feature set is contained in the previous one and contains the next. We present a dynamic-program to solve the problem efficiently for chains. We then analyze our problem formally and show that it is NP-Hard in more general settings.

Provenance Contributions Based on conversations with analysts, we identify two benefits of provenance in the feature selection process: (1) to enable *oversight* of the analysis process, which may be required for legal reasons, e.g., to make sure discriminatory features are not used during the insurance process, or for deeper understanding of how and what data is used, and (2) to *reuse* results to improve performance of COLUMBUS programs. Our approach combines existing models of workflow and data provenance [17, 19, 34]. Our proposed system captures provenance at two granularities – *coarse*, where only the inputs and outputs of COLUMBUS operations are captured, and *fine*, where the internal details of operations are captured as well. Empirically, we find that even for fine granularity capture, the runtime overhead is within 5% on real feature selection workloads. Not surprisingly, we also find that reusing results can speed up feature selection programs by orders of magnitude.

Contributions We make the following contributions:

- We take a first step towards managing the process of feature selection over data analytics systems. We organize a common set of operations for feature selection into a framework and prototype our ideas over the Oracle R environment and Python over PostgreSQL.
- We study the impact of two classical performance optimizations – batching and materialization. We formalize these optimizations in our framework and show that they can be solved efficiently in some common cases but are NP-Hard in general.
- We show two benefits of provenance in feature selection – enabling *oversight* and *reuse* of captured results for performance. We discuss the design space for granularity of provenance

capture in feature selection.

Based on our interactions with the analysts as well as Guyon and Elisseeff's popular reference on feature selection practice [30, 31], we write Columbus programs and empirically validate our contributions using real and synthetic datasets. Our performance optimizations yield speedups ranging from about 2x to 20x on these programs. We then validate our optimizations individually in detail. We show that the overhead to capture provenance is less than 5% for these programs and describe two uses for captured results.

Outline The rest of the paper is organized as follows: in Section 2, we present a motivating case study and our framework of operations. In Section 3, we discuss our system architecture and performance optimizations. In Section 4, we discuss how we manage provenance and reuse results. In Section 5, we empirically validate our performance optimizations, and provenance management. We discuss related work in Section 6, and conclude in Section 7.

2 Motivation and Preliminaries

We present a case study about a feature selection process based on our discussions with actuaries at a major insurance firm, American Family Insurance (AmFam).

2.1 Case Study: Insurance Risk Pricing

A common analytics task in the insurance domain is determining the risk of a customer in order to decide insurance pricing rates. We are collaborating with AmFam to explore the role of newer data sources in designing better insurance policies. The datasets here typically have a few hundred features from various sources (e.g., customer details and census records). Through our conversations, we learned how analysts explore the importance of each feature and decide what features to use for statistical modeling.

Analysts often use domain-specific methods for feature selection. One method they use is to split and analyze the features in separate groups (e.g., customer features and car features). They obtain descriptive statistics like correlations among features, manually choose some features, and evaluate the feature sets. Simple algorithms like stepwise selection [30–32] and Lasso are also commonly used. In between the exploration, the dataset is also filtered on features like income to focus on specific customer segments. Thus, overall, they use their insights, data manipulations, descriptive statistics, and a few popular selection algorithms. To get more accurate insights, they are looking to bring in larger datasets into such analyses, but are concerned about poor performance at scale. Higher performance on larger data can improve the turnaround times for their analyses and contribute to more informed business decisions. Also, they often have to manually manage information about the steps and results using ad-hoc notes and emails. Such information is needed to improve their analyses and to communicate their results for presentation and auditing purposes.

Discussion We also spoke to analysts in other enterprise settings – a consulting company (Deloitte), an Oracle customer, and data scientists at an e-commerce company. We found similar processes and issues for feature selection faced by the analysts. Our goal is to take a step towards managing the process of feature selection to address the two issues of performance and

High-level Task	**Example Operations**
Descriptive statistics	Mean, Variance, Correlation, Covariance
Train and score models	Linear and Logistic Regression, Cross-Validation
Automatic selection	Forward and Backward Selection, Lasso
Semi-automatic selection	Stepwise Add and Drop
Manually choose features	Insert, Remove, Combine, Create new features
Data manipulations	Select, Project, Join

Table 1: Common high-level tasks and example operations in feature selection, based on Guyon and Elisseeff [30], other literature [3, 14], and our interactions with analysts.

provenance. Based on our interactions with the analysts and the literature on feature selection practice [3, 14, 30] (see Table 1), we organize a set of common operations for feature selection into the Columbus framework. We categorize the operations in Columbus into three types – *Explore*, *Evaluate*, and *Data-Transform*. We will explain in detail shortly, but we start with an example based on our case study (see Figure 1).

Example An analyst runs Columbus operations wrapped as R functions to explore a U.S. Census dataset, which has demographic, economic, and other groups of features. A binary target feature in the dataset asks if a majority of a census block's inhabitants reply to census mails. She assesses groups of features and uses correlations to check for feature redundancies (an Evaluate operation `CorrelationX`), performs model evaluations (an Evaluate operation `EvaluateLRCV` that does cross-validation of logistic regression), selects smaller customer segments (a Data-Transform operation `Select`), combines features (an Explore operation `Union`), and runs some selection algorithms (an Explore operation `StepAdd`). She copies the printed results to text files to discuss with others and validates the results of these operations against a larger dataset. Our focus is on the two issues of performance over large data and provenance management for such feature selection programs:

- *Performance*: We study how two classical database-style performance optimizations ideas can improve performance over large data. Empirically, we show that our optimizations can yield up to 20x speedups.
- *Provenance*: We capture the provenance of such processes and show that the runtime overhead is less than 5% and needs less than 100kB of storage space.

```
1. d1 = Dataset(c("USCensus")) #Register the dataset
   #—s1 represents population-related features—#
2. s1 = FeatureSet(c("Population","NumHouses", ...))
3. l1 = CorrelationX(s1, d1) #Get mutual correlations
4. Remove(s1, "NumHouses") #Drop this feature
5. l2 = EvaluateLRCV(s1, d1) #Evaluate this set
   #—Focus on high-income areas—
6. d2 = Select(d1,"MedianIncome >= 100000")
   #—s2 represents economic features—
7. s2 = FeatureSet(c("AvgIncome","MedianIncome",...))
8. l3 = EvaluateLRCV(s2, d2)
9. s3 = Union(s1, s2) #Use both sets of features
   #—Add in one other feature—
11. s4 = StepAdd(1, s3, d1)
12. Final(s4) #Session ends with chosen features
```

Figure 1: Example snippet of a COLUMBUS program with logical operations wrapped as functions usable in R/ORE.

2.2 Operations in COLUMBUS

We start with some definitions. A *feature*, f, is a semantic signal drawn from some domain, denoted $dom(f)$. Example features include customer age, county population, etc. The *domain of discourse*, $\mathcal{F}$, is a user-given global set of uniquely-identifiable features $\{f_1, f_2, \ldots, f_n\}$, each with its own domain. The *Feature Lattice* on $\mathcal{F}$ is the transitive reduction of the partially ordered set $(2^{\mathcal{F}}, \subseteq)$. A *feature set* is an abstract data type to represent a set of features from the Feature Lattice, i.e., $F \subseteq \mathcal{F}$. A *dataset* is a relation $R(ID, f_1, \ldots, f_m)$, whose attributes form a feature set, $\{f_1, \ldots, f_m\} \subseteq \mathcal{F}$. The ID is a distinguished key attribute. A feature set $S \subseteq \mathcal{F}$ is *feature compatible* with a dataset $R(ID, f_1, \ldots, f_m)$ if and only if $S \subseteq \{f_1, \ldots, f_m\}$. We also use the notation $R(\mathcal{F})$ to denote $R(ID, f_1, \ldots, f_n)$, assuming the ID implicitly.

Table 2 summarizes the three categories of operations in COLUMBUS – *Explore* operations that traverse the Feature Lattice; *Evaluate* operations that score feature sets over the dataset; and standard *Data-Transform* operations that manipulate a dataset. Our contribution is not in defining these operations – they are already used by analysts [3,4,14,30]. However, our framework helps us tackle the two issues of performance optimizations and provenance tracking.

Explore Operations These operations enable an analyst to traverse the Feature Lattice by manipulating feature sets. We categorize them into three types – *FeatureSetOps*, *AutomaticOps*, and *SemiAutomaticOps*. FeatureSetOps are changes to feature sets that help an analyst steer the selection by manually adding or removing features. AutomaticOps are popular feature selection algorithms, which when given a feature set and a feature-compatible dataset, output another feature set. We currently consider some popular search-based algorithms like Forward and Backward selection, as well as Lasso for regression. SemiAutomaticOps are similar to AutomaticOps, but give the analyst more control. We consider Stepwise selection that can add or drop a feature one at a time based on computations over the data.

Type	Examples	Inputs	Outputs
Explore	FeatureSetOps	F_i, F_j	F_o
	AutomaticOps, SemiAutomaticOps	R_i, F_i	F_o
Evaluate	DescriptiveStatsOps, LearnerOps, ScorerOps	R_i, F_i	$D(F, V)$
Data-Transform	SPJUAQuery, Sort, Shuffle, Create	$R_i[, R_j]$	R_o

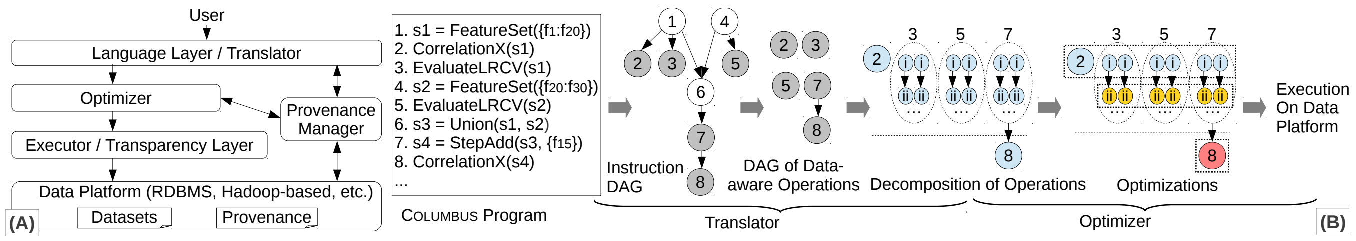
Table 2: Categorization of our operations in COLUMBUS . All F_i are feature sets ($\subseteq \mathcal{F}$), while all R_i are datasets (relations). $D(F, V)$ is a two-attribute relation wherein F is of type feature set and V is a real number.

Evaluate Operations These operations obtain various numeric scores for a feature set. We categorize them into three types – *DescriptiveStatsOps*, *LearnerOps*, and *ScorerOps*. DescriptiveStatsOps include descriptive statistics of the data, e.g., mean and variance, which take a single feature as input, and correlation and covariance, which need a pair of features. LearnerOps are a class of machine learning algorithms that compute coefficients or weights over features. This class is large, and we currently consider a few popular convex models like logistic regression, linear regression, and others that fit into the aggregation-based abstraction described by Feng et al [23]. ScorerOps score learned models over the data. We currently support prediction error, cross-validation error (e.g., of logistic regression), and Akaike Information Criterion (AIC) [30].

Data-Transform Operations These operations are common data manipulations performed by analysts to slice and dice the dataset. Recent R-based systems that have an RDBMS backend [10,48] also identify that such operations are relational and exploit the RDBMS optimizer to improve performance. Other operations include sort and shuffle, as well as simple functional transformations on features, e.g., squares and products of features.[2] Such functional transformations are managed as logical views.

Overall, our framework currently has **7** Explore, **7** Evaluate, and **4** Data-Transform operations. More details are available in our technical report [2]. Our framework is not intended to be comprehensive, but serves as a starting point since it already helps us capture real feature selection workloads that we found from our discussions with the analysts and in the literature on feature selection practice. We now discuss how we handle COLUMBUS programs and discuss our performance optimizations.

[2]For simplicity of exposition, we assume the features of the dataset are numeric.


User
Language Layer / Translator
Optimizer
Provenance Manager
Executor / Transparency Layer
Data Platform (RDBMS, Hadoop-based, etc.)
Datasets
Provenance
(A)
Columbus Program
Instruction DAG
DAG of Data-aware Operations
Decomposition of Operations
Optimizations
Execution On Data Platform
Translator
Optimizer
(B)

```
1. s1 = FeatureSet({f1:f20})
2. CorrelationX(s1)
3. EvaluateLRCV(s1)
4. s2 = FeatureSet({f20:f30})
5. EvaluateLRCV(s2)
6. s3 = Union(s1, s2)
7. s4 = StepAdd(s3, {f15})
8. CorrelationX(s4)
...
```

Figure 2: (A) High-level architecture of our system to implement COLUMBUS . (B) Illustration of the end-to-end treatment of a COLUMBUS program in our system. In the COLUMBUS program, $s1, s2, \{f_{15}\}$, etc. are feature sets.

3 Performance in COLUMBUS

We now discuss how we prototype the abstract COLUMBUS framework to be usable in the ORE environment [10]. We explain the data access patterns of COLUMBUS operations and mention how we can leverage existing optimizations. We then study how to apply batching and materialization optimizations to COLUMBUS programs.

3.1 System Overview

Our system architecture (Figure 2(A)) is based on existing R-based systems [10, 26, 48]. There are four major components: Translator, Optimizer, Executor, and Provenance Manager. The datasets reside on a Data Platform (we use an RDBMS), which is transparent to the R user [10,48]. Our novel contributions are in the performance optimizations of our Optimizer and Provenance Manager. We prototyped COLUMBUS as a library usable in an R (as well as Python) front-end for an RDBMS back-end.

Usage Settings and Examples We consider both *interactive* and *full-program* usage settings of COLUMBUS programs. In the interactive setting, an analyst runs each instruction of a program (e.g., Figure 1) one by one. In the full-program setting, she provides the full program to the system at once. COLUMBUS programs are sequences of COLUMBUS operations, with FeatureSetOps (e.g., `Union`) introducing control dependencies among operations. Executing the instructions sequentially in the full-program setting could lead to loss of opportunities to optimize performance.

We now discuss how our system executes a COLUMBUS program with an end-to-end example (Figure 2(B)). The Translator converts the program into a directed-acyclic graph (DAG) of instructions and performs static analysis to obtain a DAG of COLUMBUS operations that access the datasets (*data-aware* operations, e.g., 3, and 5 in Figure 1). The Translator then decomposes the data-aware COLUMBUS operations into workflows of *atomic* operations, which have data access patterns supported by the data platform (more details shortly). The Optimizer optimizes data accesses using our batching and materialization optimizations, and reusing optimizations in the RDBMS. It outputs job descriptions, e.g., perform a scan-based aggregation, or materialize a projection. The Executor converts the jobs output by the Optimizer into the data platform's language (e.g., SQL queries over in-RDBMS data) and obtains the results.

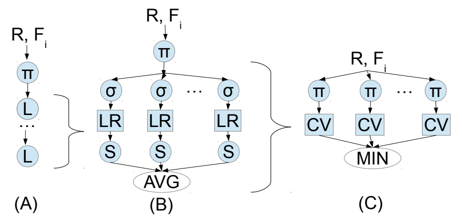


Figure 3: Decomposition of Columbus operations into workflows of atomic operations that touch the dataset. (A) CoefficientLR (B) EvaluateLRCV (C) StepAdd. Notation: σ: relational Select, π: Project, L: Learner, S: Scorer; LR and CV represent workflows.

Atomic Op.	**Data-Accesses**	**Optimized By**
Learner	Scan	Columbus Optimizer
Scorer	Scan	
SufficientStats	Scan	
Select	Scan, Index	Both
Project	Scan	
Sort & Shuffle	Sort	RDBMS Optimizer
Product & Join	Several	

Table 3: Atomic Operations and data access properties. The Columbus Optimizer applies batching and materialization to optimize the scan-based atomic operations Learner, Scorer, and SufficientStats as well as Select and Project that arise in Explore and Evaluate operations.

Atomic Operations and Data Accesses R-based systems like ORE translate operations on the primitives of R (e.g., selections on R data frames) into primitives supported by an RDBMS (e.g., relational Select) and reuse optimizations in the RDBMS [10, 12, 48]. Similarly, our Translator converts Columbus operations (e.g., `EvaluateLRCV`) into workflows of primitives, or *atomic* operations, whose data access patterns are supported by the RDBMS (or other platforms). Thus, we can reuse existing optimizations for relational operations, and study new optimizations for other operations. We start by illustrating the decomposition of data-aware Columbus operations into atomic operations:

Example 3.1. *Figure 3 shows the decomposition of three operations – CoefficientLR, EvaluateLRCV, and StepAdd. CoefficientLR (logistic regression) is a sequence of atomic Learners that are aggregations (scans). EvaluateLRCV (cross-validation of logistic regression) shuffles the data (sampling without replacement [23])*[3]*, partitions it (selections), runs CoefficientLR on each partition, runs atomic prediction error scorers (scans), and averages the scores. StepAdd is a search-based selection algorithm that takes steps through the Lattice to add one feature. It runs EvaluateLRCV (or AIC) on multiple feature sets, and picks the best set [31].*

[3] In our system, we assume that the dataset is shuffled once a priori and used for the full program.

Table 3 lists our atomic operations. Three atomic operations arise from Explore and Evaluate operations – Learner, Scorer, and SufficientStats. Each of these three is an aggregation that performs one scan of the dataset.[4]

Our focus is on applying performance optimizations *across* our atomic operations. As in existing systems [10, 48], we use the RDBMS to optimize relational operations. Our Data-Transform operations represent SQL queries that can exploit the RDBMS query optimizer. We then observe two key properties of feature selection workflows – (1) multiple operations often *scan the same dataset* (or roughly the same) independently, and (2) operations often *access only a few features* by projecting the dataset (Figure 3). We study how we can exploit these two properties to improve the performance of feature selection programs on large data. We apply two classical database-style performance optimization ideas – (1) *batching* of scan-based data accesses, and (2) *materialization* of intermediate dataset projections. To the best of our knowledge, these optimizations have not been studied in depth before for feature selection processes.

3.2 Performance Optimizations

We now discuss how we apply batching and materialization optimizations to Columbus programs. Our focus is on optimizing the data access of a given set of Columbus operations that have all their inputs (input feature sets and number of iterations) available. So we can obtain a workflow of atomic operations. If some operations depend on earlier ones for feature set inputs (e.g., node 8 in Figure 2(B)), the set of operations that occur later are optimized after their feature set inputs have been resolved at runtime .

[4]They are also algebraic aggregations [27]. Thus, it is possible to use our optimizations in data-parallel settings such as a parallel RDBMS and Hadoop-based systems as well.

3.2.1 Batching Optimization

Batching improves performance by juxtaposing the aggregation states of the independent scans and enabling them to share the same scan. For example, Figure 3(B) shows `EvaluateLRCV` running k (as in k-Fold) Learners independently, with a total of kN scans (N is the number of training iterations). Batching reduces the number of scans to N.[5] Operations that search on the Feature Lattice (e.g., `StepAdd` in Figure 3(C)) also benefit from batching. We also apply batching *across* all our scan-based operations in a COLUMBUS program (illustrated in Figure 2(B)).

We now formally state the batching problem. We are given a DAG $G(V, E)$, where V is a set of scans of the dataset, and E represents dependencies among scans. E imposes a partial order $\leq$ on V. We want to convert G into a sequence $G'(V', E')$, where each element of V' is a *batch* of scans from V obtained using a surjection $f : V \to V'$. E' imposes a total order $\leq_{E'}$ on V', and preserves the partial orders in G, i.e., $\forall u, v \in V, u \leq v \implies f(u) \leq_{E'} f(v)$. The goal is to obtain such a G' with minimum $|V'|$.

We use the following algorithm to obtain such a mapping f – sort G topologically, go over $v \in V$ in increasing topological order, and assign it to the minimum possible $v' \in V'$ that preserves the partial orders in G. The minimum possible $|V'|$ is the height of the partial order on G, and our algorithm obtains such a V' in time $O(|V| + |E|)$. The number of scans is reduced from $|V|$ to the height of the partial order on G (illustrated in Figure 4(A)). Since a scan includes not just I/O costs, but also auxiliary overheads like parsing and function calls, batching can save time even in the interactive setting when the dataset fits in memory.

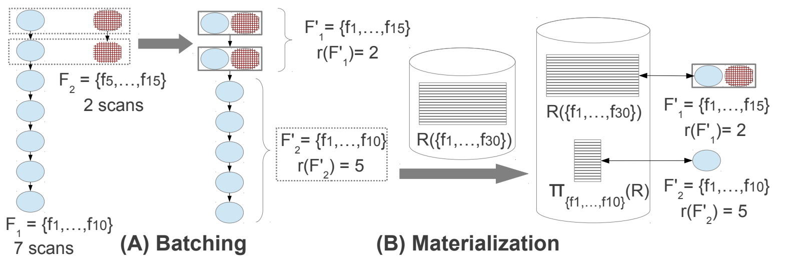
As an extension, we also formulated batching with computation costs considered. But empirically, we found this formulation's result to be less than 20% faster on real feature selection programs than our algorithm above. Since the improvement is low, we do not use this formulation for batching. We also show that batching is NP-Hard in a setting in which the system memory is a constraint (i.e., the sum of the aggregation states exceeds the available memory). The proof uses a reduction from the BINPACKING problem, and we can adopt a standard first-fit heuristic to solve our problem [24]. We do not focus on these extensions here, and present more details in our technical report [2].

3.2.2 Materialization Optimization

We start by motivating why we need a cost-based approach to optimize the materialization of projections in feature selection programs. We are given a dataset $R(\mathcal{F})$ and an operation (e.g., `CoefficientLR`) with an input feature set $F \subseteq \mathcal{F}$. Let scanning $R(\mathcal{F})$ take s units of time. Assuming all features have equal size in a dataset, scanning the materialization of F (i.e., the projected dataset $\pi_F(R)$[6]) takes only $s|F|/|\mathcal{F}|$ units. Let the time to materialize F be m units, and suppose the operation performs N scans (e.g., the number of iterations in `CoefficientLR`). So materializing F is beneficial only if $Ns > m + Ns|F|/|\mathcal{F}|$. Thus, we design a cost-based optimizer that considers costs of scans, materializations, computations, and memory. For simplicity of exposition, we use only scan and materialization costs here. More details are in our technical report [2]

[5]The partition of the dataset needed for k-Folds is done using a modulo on the serial keys (IDs) of the tuples.

[6]The ID attribute in R is implicitly retained in $\pi_F(R)$



Algorithm 1: Dynamic Program

```
//$i > 0$, use original dataset
$T(i,0) = r(F_i)\ \text{Read}(\mathcal{F}) + T(i-1,0)$
//$i > j > 0$, use the last materialization
$T(i,j) = r(F_i)\ \text{Read}(F_j) + T(i-1,j)$
//$i > 0$, materialization costs on diagonal
$T(i,i)$
 $= \min_{j=1\ldots i-1} (\text{Store}(F_j, F_i) + T(i-1,j)) + r(F_i)\ \text{Read}(F_i)$
```

Figure 4: Performance optimizations: (A) Example with two iterative scan-based operations, with input feature sets F_1 and F_2. Batching removes 2 scans and yields 2 distinct feature set accesses, $F_1' = F_1 \cup F_2$ and $F_2' = F_2$, with repetitions $r(F_1') = 2$ and $r(F_2') = 5$. (B) Note that $F_1' \supseteq F_2'$. F_2' is accessed sufficiently often, and so we materialize F_2' as a projection of the dataset relation R. Algorithm 1: Dynamic program to decide which feature set accesses to materialize.

$$\begin{aligned}
\text{Read}(F) &= \begin{cases} O(|F|), \ \textit{if} \ \textit{size}(\pi_F R(\mathcal{F})) > M \\ 0, \ \textit{otherwise} \end{cases} \\
\text{Write}(F) &= \begin{cases} O(|F|), \ \textit{if} \ \textit{size}(\pi_F R(\mathcal{F})) > M \\ 0, \ \textit{otherwise} \end{cases} \\
\text{Store}(F_1, F_2) &= \text{Read}(F_1) + \text{Write}(F_2)
\end{aligned}$$

Table 4: I/O costs for materialization optimization given a dataset $R(\mathcal{F})$. $F, F_1, F_2 \subseteq \mathcal{F}$ are feature sets. $size(\pi_F R(\mathcal{F}))$ is the table size of the materialization of F. M is the size of buffer memory.

Cost Model Given a dataset $R(\mathcal{F})$ with a full set of features $\mathcal{F}$. Given $F \subseteq \mathcal{F}$, the cost of scanning $\pi_F(R(\mathcal{F}))$ is denoted Read(F). The cost of materializing F using $R(\mathcal{F})$ is denoted Store$(\mathcal{F}, F)$ – it involves scanning $R(\mathcal{F})$ and writing $\pi_F(R(\mathcal{F}))$.[7] We can also materialize F using a materialized projection $\pi_{F'}(R(\mathcal{F}))$, if $F \subseteq F'$, whence the cost is Store(F', F). Table 4 shows our cost model in terms of only I/O costs (for simplicity of exposition). Our model accounts for both the case when a projected dataset needs to reside on disk and when it can fit in buffer memory.

We now motivate a common scenario for the materialization problem that builds upon our batching optimization. Recall that our batching algorithm produces a sequence of scans, each of which accesses a feature set that is the union of the input feature sets of the scans that were batched. Thus, successive scans only access the same feature set or smaller (see Figure 4(B)). In other terms, suppose we have two successive scans with respective input feature sets F_1 and F_2, then $F_1 \supseteq F_2$. So, if $\pi_{F_1}(R)$ is materialized, it can potentially be used for both scans. We now

[7]For any $F \subseteq \mathcal{F}$, Store$(F, F) = 0$.

formally state our materialization optimization problem.

Problem Statement Fix dataset $R(\mathcal{F})$. We are given a set of accesses of subsets of $\mathcal{F}$, labeled $\mathcal{S} = \{F_1, \ldots, F_N\}$, where $F_i \subseteq \mathcal{F}, \forall i = 1 \ldots N$. The subsets are accessed with *repetitions* $\{r(F_i)\}_{i=1}^{N}$, i.e., F_j is accessed $r(F_j)$ times. A *materialization plan* is a subset $\mathcal{S}' \subseteq \mathcal{S}$ whose feature sets are materialized and used for data accesses. We want to obtain a materialization plan with minimum cost. We formulate it as an optimization problem:

MatOpt: $\min_{\mathcal{S}' \subseteq \mathcal{S}}$ $\text{Cost}(\mathcal{S}')$,

$$\begin{aligned}\text{where } \text{Cost}(\mathcal{S}') = & \sum_{F \in \mathcal{S}'} \min_{F' \in \mathcal{S}' \cup \{\mathcal{F}\} \setminus \{F\}:\ F' \supseteq F} \text{Store}(F', F) \\ & + \sum_{F \in \mathcal{S}} r(F) \min_{F' \in \mathcal{S}' \cup \{\mathcal{F}\}:\ F' \supseteq F} \text{Read}(F')\end{aligned}$$

In other words, we materialize the feature sets in $\mathcal{S}'$ and use the "thinnest" among them (or $R(\mathcal{F})$) to serve each feature set. Our focus is on MatOpt when $\mathcal{S}$ is a *chain* with respect to set containment, i.e., $\mathcal{F} \supseteq F_1 \supseteq F_2 \cdots \supseteq F_N$. Apart from the output of our batching optimization, the chain property can also arise in other feature selection scenarios, for example, during stepwise search on the Feature Lattice. We now present an algorithm to obtain an optimal solution to MatOpt efficiently for chains. Our idea is to exploit the fact that we will only use the thinnest feature-compatible materialized dataset to serve each feature set. Thus, we explore the space of materialization choices from F_1 up to F_N incrementally and track the best decisions using a dynamic programming-style approach.

Dynamic Program We obtain $\mathcal{S}^*$ using a dynamic program that computes cumulative costs of materializations recursively (Algorithm 1). We define a cost $T(i,j)$ as follows: we count the cumulative cost of the best materialization plan up to "slot" i, counting from slots 1 to N (slot i corresponds to the plan to serve feature sets $F_1, \ldots, F_i$). The entry j represents the slot at which the latest materialization was done, i.e., F_j was materialized. Thus, fixing j, we will use $\pi_{F_j}(R)$ to serve feature sets $F_k, k > j$. The entries on the diagonal (i.e., $i = j$) count the cost of materializing F_i and serving F_i using $\pi_{F_i}(R)$. The costs are memoized using an $N \times N$ memo. We obtain the optimal plan at slot N (i.e., serving up to F_N). The lowest cost $\mathcal{S}^*$ is obtained by maintaining backpointers in the memo.

Theorem 3.1. *Algorithm 1 solves* MatOpt *for chains in time $O(N^2)$ with space $O(N^2)$.*

3.2.3 Extensions and Further Analyses

We now analyze more general settings for MatOpt and explain some theoretical challenges that arise.

Constant-width Lattice We first consider a generalizations of chains when $\mathcal{S}$ is a lattice of a constant width. More precisely, $\mathcal{S} = \{F(i,j) | 1 \leq i \leq N, 1 \leq j \leq k\}$, where k is a given constant. We are also given $F(i, j_1) \supseteq F(i+1, j_2), \forall i \geq 1, \forall j_1, j_2$. We can extend Algorithm 1 to have k entries per slot in the memo and recurse over all k. We show that computing an optimal is in time polynomial in N, but exponential in the constant k.

Theorem 3.2. MatOpt *for constant-width lattices can be solved in time* $O(k^2 2^k N^{k+1})$ *with space* $O(N^{k+1})$.

General Lattice The above analysis hints that the width of the lattice affects the hardness of MatOpt. In fact, we can show that, unless $\mathsf{P} = \mathsf{NP}$, no fully polynomial time algorithm can solve MatOpt.

Theorem 3.3. MatOpt *is* NP*-Hard in the width of the lattice on* $\mathcal{S}$.

Our proof uses a reduction from the NP-Hard problem SetCover [24]. Note that our batching optimization mitigates this issue by constructing chains of input feature sets as input to MatOpt (i.e., width $k = 1$, and thus $O(N^2)$).

We give more detailed analyses and proofs in our technical report [2]. We also analyze our problem in other general scenarios. One scenario we discuss in more detail is online optimization, when not all repetitions are known a priori (say, due to data-dependent number of iterations). We discuss an online algorithm that decides materializations dynamically when the repetitions are unknown, and guarantees a 2-approximation to the optimum.

4 Provenance in Columbus

We describe why we capture provenance of the feature selection process in Columbus , and briefly describe the design space and implementation details.

4.1 Uses of Provenance in Columbus

In Columbus , we capture provenance for two principal reasons: (1) to enable *oversight* of the analysis process and (2) to enable *reuse* of results to improve performance.

Oversight Provenance can be used for simple governance and deeper analysis. For instance, in insurance, an analyst cannot use discriminatory features, but exactly what features are considered discriminatory differs between jurisdictions. For example, credit score is considered discriminatory in California, but not in Wisconsin. Thus, existing commercial analytics systems already capture such governance information for auditing purposes [9, 13].

From our conversations with analysts, we learned that analysts and their managers often want to query more detailed information about the feature selection process. One kind of query deals with the inputs and outputs of feature selection operations to compare results. For example, "Which feature set evaluated has the lowest cross-validation error?" and "Which feature pairs have correlation greater than 0.7?" Another kind of queries probe the internals of feature selection operations to understand the results more deeply. For example, "Which feature sets were visited by stepwise selection?" and "Was a given feature used by any operation?" Such provenance information is often managed manually, e.g., in emails, spreadsheets and handwritten notes, which makes it difficult to query. A goal of Columbus is to automate the capture of provenance for this information.

Reuse of Results We also observed that during a feature selection process, computations might be repeated (even by the same analysis). For example, an analyst might repeat an operation during her exploration or an operation might have been executed as part of another operation (e.g., `EvaluateLRCV` as part of `StepAdd` as in Figure 3(C)). By capturing the results of COLUMBUS operations, we can detect opportunities to reuse results to improve performance.

4.2 Coarse and Fine Provenance in Columbus

COLUMBUS builds on existing models of workflow and data provenance [17, 19, 34]. Specifically, we capture workflow provenance of Explore and Evaluate operations at multiple granularities. We explore two extreme granularities in our prototype: (1) a *coarse* granularity, in which we capture only the inputs and outputs of individual operations and (2) a *fine* granularity, in which we capture internal details of Explore and Evaluate operations. For example, at the coarse granularity, we store only the inputs and outputs (e.g., feature sets) of a `StepAdd`. At the fine granularity, we capture the inputs and outputs of the operations that constitute a `StepAdd`, namely `EvaluateLRCV` and `CoefficientLR` as well (e.g., the coefficents and scores obtained as in Figure 3(C)). Capturing coarse granularity of provenance enables queries over the inputs and outputs. Fine granularity enables queries over the internals of COLUMBUS operations, but potentially at a higher runtime and storage overhead. For example, a `StepAdd` might explore a large number of feature sets, all of which need to be captured in the fine granularity setting. However, the fine granularity setting also enables more opportunities for reuse than the coarse setting. We use the fine granularity setting as the default, but let an analyst choose the granularity of provenance capture for her application.

Detecting Reuse Opportunities To detect opportunities for reuse of results, we represent a COLUMBUS program as an *expression tree* of operations. We perform subexpression matching to find syntactic matches for expressions with only Explore and Evaluate operations. Our approach of subexpression matching is sound, but not complete. That is, we only find correct reuse opportunities, but we may miss some logically equivalent reuse opportunities. This design decision was motivated by the following three factors: (1) our operations are *functional*, i.e., given the same input, they produce the same output, (2) subexpression matching captures the common cases of reuse opportunities that we observed in real feature selection workloads, and (3) our data expressions (Data-Transform operations) are arbitrary SQL queries (with bag semantics) and aggregation. For such queries, deeper containment relationships are still theoretically open questions [36]. After detecting reuse opportunities, we perform subexpression elimination and send the reduced COLUMBUS program to the Optimizer.

5 Experiments

We present an empirical evaluation of our contributions on real feature selection processes and datasets. We validate that our performance optimizations are able to achieve significant speedups on various scales of datasets. We then measure the overhead and the benefits of provenance in answering queries without recomputations.

Dataset	Features	Tuples	Size
FordDrivers	30	604K	143MB
USCensus	161	109K	134MB
SynFord	30	35M-70M	8GB-16GB
SynCensus	161	7M-14M	8GB-16GB

Table 5: Dataset Statistics.

Datasets and Tasks We use two publicly available datasets, FordDrivers and USCensus, from the Kaggle Competition website.[8] USCensus is a dataset for the task of predicting mail responsiveness of people in different Census blocks. FordDrivers is a dataset from Ford for predicting the alertness of a car's driver. Both tasks are modeled as binary classification problems. We chose these datasets since they are public but are still representative of the datasets that we found to be typical at AmFam and other enterprise settings, namely, data with dozens to hundreds of features. Each feature has a semantic interpretation that an analyst uses in the feature selection process. In order to measure the impact of our performance optimizations on larger data, we synthesize larger datasets that maintain the schema of the above two datasets. Table 5 presents the dataset statistics.

Columbus Programs We write four Columbus programs based on our interactions with analysts at AmFam and the literature on feature selection practice [30]. We capture two different kinds of feature selection workloads – the first involves the analyst controlling the feature selection process using her domain knowledge and hence, fewer selection algorithms. The second involves more usage of selection algorithms. Specifically, programs Ford1 (on FordDrivers) and Cens1 (on USCensus), which represent the first kind, split the features into 3 groups, and analyze each group separately using more Evaluate operations (specifically `EvaluateLRCV`). Ford2 and Cens2 use more Explore operations (specifically `StepDrop`). We present statistics about the programs in Table 6, but due to space constraints, list the full programs in our technical report [2]. Since Ford1 and Cens1 access independent groups of features, they have up to 60 independent data accesses and almost no dependencies across operations. Ford2 and Cens2 first select less than one-fifth of the features automatically and then reduce the feature set sizes using `StepDrop`. Hence, these two have only up to 20 independent data accesses, but have several iterations accessing less than one-fifth (and up to just one-tenth) of the features.

Experimental Setup All our experiments are run on an identical configuration: machines with AMD Opteron 2212 HE CPUs (4 cores), 8GB of RAM, and 200GB disk. The kernel is Linux 2.6.18-238.12.1.el5. Each runtime reported is the average of three warm-cache runs. Our prototype over ORE uses Java user-defined functions to implement our atomic scan-based operations, while our prototype over PostgreSQL uses C user-defined functions. Our focus is on measuring the relative performance gains achieved by our optimizations. Our results do not reflect the absolute performance of ORE. Thus, we use XDB and YDB to refer to our prototypes anonymously in the results.

[8] `http://www.kaggle.com`

Program	**Ford1**	**Ford2**	**Cens1**	**Cens2**
# Evaluate Ops	10	3	10	3
# FeatureSetOps	14	9	17	9
# Other Explore Ops	2	4	1	4
Total # Instructions	26	16	28	16

Table 6: Program Statistics: Evaluate operations include DescriptiveStatsOps, `CoefficientLR`, and `EvaluateLRCV`. FeatureSet operations are manual and do not touch the dataset. Other Explore operations here are automatic and stepwise selection algorithms.

5.1 End-to-End Overview

We first present end-to-end overviews of executing our programs. Due to space constraints, we only discuss Ford1 and then Cens2 here. We found similar results on the other programs and present them in the technical report [2].

Interactive Setting Figure 5 shows the total runtimes of instructions in Ford1. The two strategies compared are: naive execution without any optimizations (NoOpt) and applying the batching optimization (BatOpt). The data fits in memory here, and so we only consider batching within each COLUMBUS operation. We expect to see minor speedups since I/O is not involved.

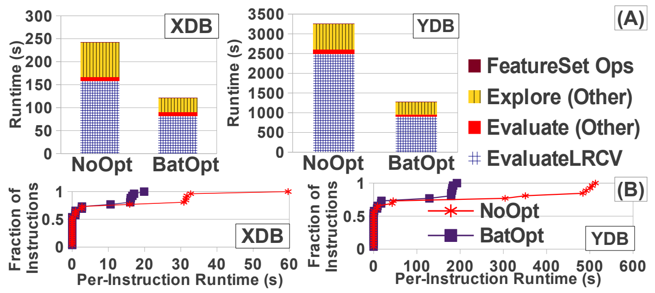


Figure 5: Overview of the interactive setting for Ford1 on the Ford dataset on both systems. NoOpt is the case of naive execution without any batch optimizations, and BatOpt is when batching is performed (for `EvaluateLRCV`, `StepAdd`, and `StepDrop`). (A) Breakup of total runtime into individual classes of operations. (B) CDF of per-instruction runtimes.

Batching speeds up Ford1 by 2x in both systems. The runtime is dominated by `EvaluateLRCV` and stepwise selection (`StepAdd` and `StepDrop`), and these operations are sped up by batching since data access overheads like parsing and function calls are reduced. The CDF of per-operation runtimes shows that over half the operations in Ford1 finish in under 1s – these are the FeatureSetOps that do not access the dataset.

Full-Program Setting Scalability We now study the impact of our optimizations across full programs on larger data scales, where the I/O becomes significant. We plot the runtimes of Ford1 and Cens2 against the dataset size for three settings – NoOpt, BatOpt, and with our materialization

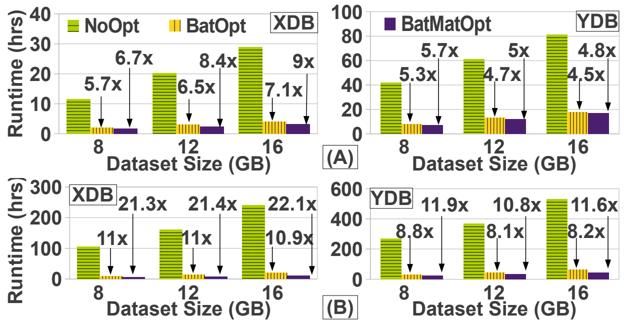


Figure 6: Full-program setting: (A) Ford1 (B) Cens2 on both systems as the dataset size is scaled up. NoOpt is naive execution, BatOpt applies batching, and BatMatOpt applies both batching and materialization optimizations. The speedups achieved by BatOpt and BatMatOpt against NoOpt are also shown.

optimization applied after batching (BatMatOpt). Since Ford1 has many independent accesses to the dataset but accesses most of the features, we expect significant savings primarily from batching alone. But since Cens2 has dependencies across operations and accesses fewer features, we expect that the materialization optimization will also result in significant savings in addition to batching. Figure 14 shows the results (we observe similar results for the other two programs).

We see that overall, our optimizations improve performance across the board. Batching alone yields speedups of up to 7x on Ford1, since it saves scans both within and across COLUMBUS operations. The speedups in PYPG are lower for smaller datasets because the relative cost of I/O against computation is lower since the RDBMS caches larger portions of the datasets. On RORA, the speedups vary less since non-I/O data access overheads dominate the runtime. In addition to batching, performing our materialization optimization increases the speedups to 9x on PYPG. However, the savings on RORA are smaller since non-I/O data access overheads dominate scan savings. On Cens2, we see that batching alone yields 11x speedups on PYPG on 8x on RORA. However, since Cens2 has more repeated accesses of small feature sets, using materialization optimization in addition to batching improves the speedup on both systems. The speedup is up to 22x on PYPG and 12x on RORA. Overall, we see that there are scenarios in such programs where our optimzations can yield speedups of over an order of magnitude. We now perform a detailed study of our performance optimizations to evaluate their impact more precisely.

5.2 Performance Optimizations in Detail

We now study our performance optimizations in detail using a synthetic dataset SynFord (70M rows, 16GB). We observed similar results on the other synthetic datasets and skip their results here due to space constraints.

Batching Optimization First, we study the impact of the batching optimization by validating that batching can speed up `EvaluateLRCV`. The number of folds is set to 10. We measure the runtime of one training iteration of `EvaluateLRCV` without any batching (NoOpt), and after applying batching (BatOpt). Figure 16 plots the results.

We see that batching speeds up `EvaluateLRCV` by up to 7x on PYPG and 5.7x on RORA. We do not see 10x speedup for 10 folds because batching saves only scan times, not computation times.

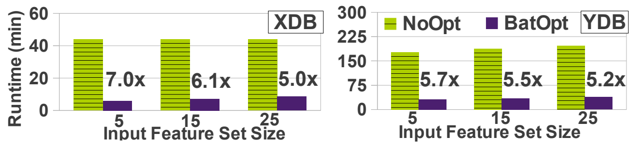
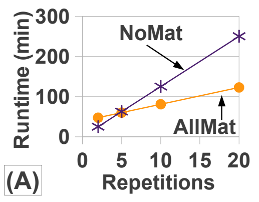


Figure 7: Runtimes of `EvaluateLRCV` (one training iteration, 10-fold CV). The respective speedups BatOpt against NoOpt are shown above the bars.

For an input set of size 5 for NoOpt in PyPG, we verified that the scan time is about 7x the computation time. Thus, after we batch all folds, the computation times dominate, resulting in a 7x speedup over NoOpt. The speedup obtained by batching is higher for smaller input set size since the relative computation times are smaller. We also verified that batching speeds up operations like `StepAdd` where multiple feature sets are batched. For the same input set sizes as in Figure 16, we observed speedups of 2x to 9x [2].

Materialization Optimization We now verify that materialization involves optimizing a performance tradeoff and that our optimizer can pick optimal plans. We compare our optimizer's plan (based on the dynamic program in Section 3.2.2) against two naive strategies – not materializing anything (NoMat) and materializing all projections (AllMat). We illustrate the tradeoff by varying the values of repetitions. The input we consider here is a chain of three feature sets of lengths 5, 10, and 15 (out of 30 features) for I/O-bound operations (`CoefficientLR`). We fix all three repetitions to be equal, and show the runtimes in Figure 8.[9]



Repetition	Speedup of Optimal	
	v NoMat	v AllMat
2	1x	1.9x
5	1.4x	1.3x
10	1.7x	1.1x
20	2.1x	1x

Figure 8: (A) Runtimes of NoMat vs AllMat for different repetition settings for a set of three `CoefficientLR` operations. The table shows speedups of the optimal materialization strategy against both NoMat and AllMat.

Figure 8(A) shows that neither NoMat nor AllMat is always better – it depends on the repetition value. This motivates our cost-based optimization approach, which takes repetitions into account in order to obtain a materialization plan. The table in Figure 8 shows the speedups achieved by our optimizer's plan against both the naive strategies. When the repetition is low (e.g., 2), the better strategy is to not materialize anything, and our optimal strategy matches NoMat. But when the repetition is high (e.g., 20), the better strategy is to materialize all the feature sets and our optimal strategy matches AllMat. For intermediate values of repetition, our optimal strategy chooses to materialize some (not all) of the feature sets, and finds a plan faster than both NoMat

[9]These numbers are on PyPG, which exhibits materialization trends clearly for exposition sake.

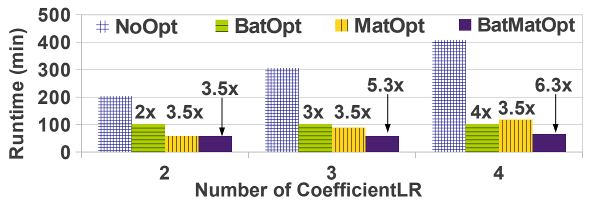


Figure 9: Role of the number of operations in the impact of our performance optimizations. We consider up to 4 `CoefficientLR` operations here. The speedups of BatOpt, MatOpt, and BatMatOpt against NoOpt are also shown.

and AllMat. We also validated that our cost model is able to capture the scan and materialization times accurately to enable such optimization. More details are discussed in our technical report [2].

Sensitivity to Program Size Finally, we study the role of the number of operations in the program on the impact of our optimizations. The input we consider here is again a set of `CoefficientLR` operations. The operations all have equal input feature set sizes (5), any pairwise unions results in 10 features. The number of iterations is 25 for each. We plot the runtimes of four strategies – naive execution without any of our optimizations (NoOpt), batching alone (BatOpt), materialization optimization alone (MatOpt), and both batching and materialization optimization (BatMatOpt). Figure 9 plots the runtimes and also shows the speedups achieved by the other strategies against NoOpt.

We see that NoOpt grows proportionately with the number of operations. BatOpt's runtime is largely flat because these operations are all scan-bound, even after batching. Thus, we see linear speedups here. MatOpt chose to materialize everything, and achieves a speedup of 3.5x. It is independent of the number of operations since the same speedup applies to each operation. BatMatOpt, which applies both optimizations, is faster than each individually. It achieves a speedup of 6.3x for 4 operations here by combining the benefits of both batching and materialization.

5.3 Provenance Management

We now present the storage and runtime overhead for capturing provenance at each of the two granularities – coarse and fine. Table 7 lists the results for all four programs (run on their respective datasets).

We see that the storage overhead is small for capturing results at coarse granularity – it only stores the inputs and outputs of Columbus operations and so the overhead is only in the order of kilobytes. Storing at fine granularity (internal details of the operations) requires space in the tens of kilobytes. The runtime overhead of capturing provenance at coarse granularity is only 0.1% and that of fine granularity is up to 5%. Thus, the runtime is still dominated by computations on the data. Cens1 and Cens2 have higher overhead (space and runtime) since USCensus has more features, which means more information is captured.

Next, we measure the utility of captured results in answering provenance queries. Without provenance information, we might have to recompute some operations of the program. Table 8 presents the results for three example queries. Note that Q1 and Q3 are about an individual session, while Q2 compares two sessions. We see that reusing captured results (with provenance)

Provenance Granularity	Program/Dataset			
	Ford1	Ford2	Cens1	Cens2
Storage Overhead (kB)				
Coarse	2.1	5.2	4.4	11
Fine	30	58	43	210
Capture Overhead (Extra % Runtime)				
Coarse	< 0.1%	< 0.1%	0.1%	0.1%
Fine	0.5%	1%	4.6%	3.2%

Table 7: Overhead for capturing provenance.

Query	Ford1 + Ford2		Cens1 + Cens2	
	Recomp.	With Prov.	Recomp.	With Prov.
Q1	234	< 0.1	66	< 0.1
Q2	585	0.2	178	0.1
Q3	85	< 0.1	32	0.1

Table 8: Provenance Queries. Q1: "Which feature set has lowest score from EvaluateLRCV?". Q2: "Which feature sets were input to EvaluateLRCV in both sessions?". Q3: "Which feature sets were visited in stepwise selection?". "Recomp." is with recomputations, while "With Prov." reuses captured results. The runtimes are in seconds.

avoids the need to recompute and since we currently query provenance in memory, query answering times are low.

In summary, we discussed examples of feature selection processes, and showed how our framework can help an analyst capture them using COLUMBUS operations. We validated that two performance optimizations – batching and materialization – can yield significant performance benefits over large-scale data, without the analyst having to hand-tune the code. We measured the overhead of managing provenance in our system and showed how we can answer provenance queries and reuse results.

6 Related Work

Feature Selection Algorithms Algorithms for feature selection have been studied in the statistical and machine learning literature for decades [20, 30–32, 37]. A typical formalization is to obtain one subset of the features of a given dataset, subject to some optimization criteria. Feature selection algorithms are categorized into *Filters*, *Wrappers*, and *Embedded* methods. Filters assign scores to features independent of what statistical model the features are used for. Wrappers are meta-algorithms that score feature sets with a statistical model, primarily using heuristic search. Embedded methods wire feature selection into a statistical model [31]. Our conversations with analysts revealed that feature selection is often a data-rich process with the analyst in the loop, not a one-shot algorithm. Our goal is to take a step towards managing the process of feature selection using data management ideas. We aim to leverage popular selection algorithms, not design new ones.

Scaling Feature Selection Scaling individual feature selection algorithms to larger data has received attention in the past for specific platforms. Oracle Data Mining offers three popular feature selection algorithms over in-RDBMS data [9]. Singh et al. parallelize forward selection for logistic regression on MapReduce/Hadoop [47]. In contrast, our focus is on building a generic framework for feature selection processes, rather than specific algorithms and platforms. So, we can also adopt their techniques for our operations.

Analytics Systems Systems that deal with data management for statistical and machine learning techniques have been developed in both industry and academia. These include data mining toolkits from major RDBMS vendors, which integrate specific algorithms with an RDBMS [9, 33]. Similar efforts exist for other data platforms [1]. More recently, R-based analytics systems have gained popularity [10, 12, 26, 48]. Our work focuses on the data management issues in the process of feature selection, and our ideas can be integrated into these systems. There has also been a massive research push towards identifying higher-level abstractions in analytics as “sweet-spots” between the two streams of specific algorithms and a Turing-complete language, akin to the decoupling in relational systems. Examples include Bismarck [23], GraphLab [40], MLBase [39], ScalOps [21], and Tuffy [42]. Our work is inspired by such efforts, but to the best of our knowledge, ours is the first work that aims to apply such ideas to managing feature selection [18].

Performance Optimizations Batching is a classical performance optimization idea [45, 49], and batching mechanisms have been studied in recent projects as well [25, 43, 49]. Our contribution is in identifying that batching is critical to improve performance in feature selection settings. We can leverage existing batching mechanisms to implement our batching optimization for feature selection. Our batching formulation also considers memory footprints of operations in addition to performance costs. Automatically selecting materialized views is a classical problem in relational query optimization [16, 29, 41, 46]. The problem of choosing an optimal set of views to materialize is motivated by Gupta et al [29]. But our problem focuses specifically on projection views, which we identify as being common in feature selection settings. We present analyses of our problem and show an important special case where our problem can be solved optimally in polynomial time. Our focus on performance optimizations across full programs was inspired by similar efforts in RIOT-DB [48] and SystemML [26]. RIOT-DB optimizes I/O by rearranging page accesses for specific loop constructs in an R program [48]. The optimizations that we focus on – batching and materialization – are orthogonal to theirs. Our framework also enables us to apply our optimizations automatically across an entire COLUMBUS program. We leave it as future work to see if our optimizations can be applied to general R code. SystemML [26] converts R-style programs to workflows of MapReduce jobs. They describe an optimization called piggybacking, which enables sharing of data access by jobs that follow each other. In contrast, our batching optimization fuses the data accesses of independent jobs. We also study materialization optimization, and demonstrate non-obvious tradeoffs that arise.

Provenance There exists a lot of work in the database literature on provenance models [22, 28, 34], querying provenance [38, 44], and systems for provenance [8, 15, 17, 35]. Our work applies recent provenance research ideas to manage provenance in feature selection processes. We also discuss how we can detect opportunities to reuse results captured as provenance to improve performance.

7 Conclusion and Future Work

Feature selection is a key step in analytics applications. Increasingly, larger datasets are used in feature selection applications. Recent R-based data processing systems aim to address scalability needs, but the level of abstraction at which analysts think of feature selection is not present in current R-based systems. We propose to use a set of common feature selection operations in a framework that we name COLUMBUS . We prototype COLUMBUS over two analytics environments, and study two classical ideas for optimizing performance: batching and materialization. We also show how we can manage and exploit the provenance of feature selection processes for both oversight and improving performance.

Our immediate future work is to continue our investigation into the problem of feature engineering in data analytics systems. We are investigating both how to bring less structured sources of data into the analysis problem, e.g., data from text or physical sensors and techniques to improve the process, e.g., speculative execution of code and feature recommendations in our system to aid the analyst.

8 Acknowledgements

We thank Vaishnavi Sashikanth (Oracle), American Family Insurance, and Jim Guszcza (Deloitte/UW-Madison) for invaluable discussions.

References

[1] Apache Mahout. `mahout.apache.org`.

[2] COLUMBUS: Feature Selection on Data Analytics Systems. `hazy.cs.wisc.edu/hazy/papers/fexsel.pdf`.

[3] Feature Selection and Dimension Reduction Techniques in SAS. `nesug.org/Proceedings/nesug11/sa/sa08.pdf`.

[4] FSelector Package in R. `cran.r-project.org/web/packages/FSelector/index.html`.

[5] Gartner Report on Analytics. `gartner.com/it/page.jsp?id=1971516`.

[6] IBM Report on Big Data. `www-01.ibm.com/software/data/bigdata/`.

[7] IDC Report on Big Data. `idc.com/getdoc.jsp?containerId=232186#.UMIO_FWRxx4`.

[8] Kepler Provenance Framework. `kepler-project.org`.

[9] Oracle Data Mining. `oracle.com/technetwork/database/options/advanced-analytics/odm`.

[10] Oracle R Enterprise. `docs.oracle.com/cd/E27988_01/doc/doc.112/e26499.pdf`.

[11] Project R. `r-project.org`.

[12] SAP HANA and R. `help.sap.com/hana/hana_dev_r_emb_en.pdf`.

[13] SAS Report on Analytics. `sas.com/reg/wp/corp/23876`.

[14] Variable Selection in the Credit Card Industry. `nesug.org/proceedings/nesug06/an/da23.pdf`.

[15] VisTrails System. `vistrails.org`.

[16] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[17] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, Dec. 2011.

[18] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[19] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008*, pages 1072–1081, 2008.

[20] D. E. Boyce, A. Farhi, and R. Weischedel. *Optimal Subset Selection.* New York: Springer-Verlag, 1974.

[21] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160, 2012.

[22] P. Buneman and W.-C. Tan. Provenance in Databases. In *SIGMOD*, 2007.

[23] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, 2012.

[24] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[25] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig experience. In *PVLDB*, 2009.

[26] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[27] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[28] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[29] H. Gupta and I. Mumick. Selection of views to materialize in a data warehouse. *IEEE TKDE*, 17(1):24–43, Jan. 2005.

[30] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 3:1157–1182, Mar. 2003.

[31] I. Guyon, S. Gunn, M. Nikravesh, and L. A. Zadeh. *Feature Extraction: Foundations and Applications.* New York: Springer-Verlag, 2001.

[32] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data mining, inference, and prediction.* New York: Springer-Verlag, 2001.

[33] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. In *PVLDB*, 2012.

[34] R. Ikeda, A. D. Sarma, and J. Widom. Logical Provenance in Data-Oriented Workflows. In *ICDE*, 2013.

[35] R. Ikeda and J. Widom. Panda: A System for Provenance and Data. In *TAPP*. USENIX Association, 2010.

[36] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for real conjunctive queries with inequalities. In *PODS*, 2006.

[37] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant Features and the Subset Selection Problem. In *ICML*, pages 121–129, 1994.

[38] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying Data Provenance. In *SIGMOD*, 2010.

[39] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[40] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*, 2010.

[41] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Rec.*, 41(1):20–29, Apr. 2012.

[42] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. *PVLDB*, 4(6):373–384, 2011.

[43] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1-2):494–505, Sept. 2010.

[44] A. D. Sarma, A. Jain, and D. Srivastava. I4E: Interactive Investigation of Iterative Information Extraction. In *SIGMOD*, 2010.

[45] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[46] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.

[47] S. Singh, J. Kubica, S. Larsen, and D. Sorokina. Parallel Large Scale Feature Selection for Logistic Regression. In *SDM*, 2009.

[48] Y. Zhang, W. Zhang, and J. Yang. I/O-Efficient Statistical Computing with RIOT. In *ICDE*, 2010.

[49] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.

A Operations in Columbus

As discussed in Section refsec:system, the operations can be categorized as Explore, Evaluate and Data-Transform operations. In the following, we shall eloborate on the operations in each of the categories.

A.1 Explore Operations

These operations enable an analyst to traverse a lattice.

A.1.1 Feature Set Operations

Feature Set Union:

- *Input*:
  - Feature set relation $F_{in1}(f)$, $F_{in2}(f)$
- *Output* : Feature set relation $F_{out}(f)$
  - Feature set relation $F_{out}(f)$
- *Operation* : t_{in1} and t_{in2} denote the set value in the input relations, and let t_{out} denote the set value in the output relation. Then $t_{out} = t_{in1} \cup t_{in2}$.
- *Condition*:
  - If F_D is the set of attributes of the data/feature vector relation. Then $t_{in1} \subseteq F_D, t_{in2} \subseteq F_D$ and $t_{out} \subseteq F_D$.

Feature set Intersection:

- *Input*:
  - Feature set relation $F_{in1}(f)$, $F_{in2}(f)$
- *Output* :
  - Feature set relation $F_{out}(f)$
- *Operation* : t_{in1} and t_{in2} denote the set value in the input relations, and let t_{out} denote the set value in the output relation. Then $t_{out} = t_{in1} \cap t_{in2}$.
- *Condition*:
  - If F_D is the set of attributes of the data/feature vector relation. Then $t_{in1} \subseteq F_D, t_{in2} \subseteq F_D$ and $t_{out} \subseteq F_D$.

Feature set Difference:

- *Input*:
  - Feature set relation $F_{in1}(f)$, $F_{in2}(f)$
- *Output* :
  - Feature set relation $F_{out}(f)$
- *Operation* : t_{in1} and t_{in2} denote the set value in the input relations, and let t_{out} denote the set value in the output relation. Then $t_{out} = t_{in1} - t_{in2}$.
- *Condition*:
  - If F_D is the set of attributes of the data/feature vector relation. Then $t_{in1} \subseteq F_D, t_{in2} \subseteq F_D$ and $t_{out} \subseteq F_D$.

A.1.2 Automatic operations

Exploratory Operations : Automatic (Lasso):

- *Input*:

    - Dataset relation $D(F_D, ID)$
- *Output* :
    - Feature set relation F_{out}
- *Condition*:
    - If F_D is the set of attributes of the data/feature vector relation. Let t_{out} denote the tuple in F_{out}. Then $t_{out} \subseteq F_D$.

A.2 Evaluate Operations

A.2.1 CoefficientLearner

The operations compute coefficients (or weights) over the features, given a dataset. An example of CoefficientLearner is `CoefficientLR`, which computes the coefficents for the technique of logistic regression. This operations is given a input feature set, input state (the initial values of the coefficents), learner-specific parameters. It returns the output state (the learned values of the coefficents).

- *Input*:
    - Coef Parameters $\{P\}$
    - Data $D(F_D, ID)$
    - Initial state $S_{in}(FV_{in} = (F_{in}, I_{in}))$, where $F_{in} \subseteq F_D$. The state contains initial values of coefficents.
- *Output* :
    - Output State $S_{out} = (F_{out}, I_{out}))$
- *Condition* :
    - $F_{in} = F_{out}$

Scorer The scorers compute standard scores over the dataset, given learned coefficents. Examples of scorers include *prediction error* (say, classification error for logistic regression) and cross-validation error (discussed next). We are given an input state (of coefficient) values and input parameters and the operation returns a numeric score. It is formally represented as, $S = Op_{scorer}(S_{in}, \{P\}, D)$. $S \in \mathbf{R}$. S_{in} represent the input state(coefficient values) and $\{P\}$ represent the input parameters (for instance, the scoring method to used) and D represents the dataset relation.

- Input : $S_{in}, \{P\}, D$
- Output: S

EvaluateLRCV EvaluateLRCV is a scorer that computes the cross-validation error and is a workflow of learners and scorers. It is formally denoted as $S = Op_{evallrcv}(F_{in}, \{P\}, D)$, where F_{in} represents the input feature set and $\{P\}$ represent the input parameters (for instance the number of folds) and D represent the input dataset relation. It is an work flow of atomic operations and can be denoted as the the composition of atomic operators for one fold is given as $Op_{scorer} \circ Op_{learner} \circ \sigma \circ \pi(F_{in}, D, \{P\})$. [10]

In the case of multiple folds, the output scores are averaged and the result is returned.

- Input : F_{in}, $\{P\}$, D
- Output: S
- Structure: $Op_{scorer} \circ Op_{learner} \circ \sigma \circ \rightleftharpoons \circ \pi(F_{in}, D, \{P\})$

EvaluateAIC EvaluateAIC is similar to EvaluateLRCV, where given a feature set F_{in}, $\{P\}$, and dataset relation D, it computes the Akaike Information Criterion (AIC) score. It is denoted as follows $S = Op_{evalaic}(F_{in}, \{P\}, D)$, where F_{in} represents the input feature set and $\{P\}$ represent the input parameters and D represent the input dataset relation. It is a workflow of atomic operation and can be denoted as a sequence of atomic operations. It can be represented as $S = Op_{Scorer} \circ Op_{learner}(F_{in}, \{P\}, D)$.

- Input : F_{in}, $\{P\}$, D
- Output: S
- Structure : $S = Op_{Scorer} \circ Op_{learner}(F_{in}, \{P\}, D)$.

Correlation Correlation operation takes 2 input features F_1, F_2 and returns a correlation value C, it is represented as $C = Op_{corr}(F_1, F_2, \{P\}, D)$

SemiAutomatic operations **StepAdd/Drop**

- Input : F_{in}, D, $\{P\}$
- Ouput : F_{out}.
- Condition : `StepAdd`: $|F_{out}| = 1 + |F_{in}|$
  `StepDrop`: $|F_{out}| = |F_{in}| - 1$.
- Structure : $argmin_{F_s}($ `EvaluateLRCV`$(F_s, D, \{P\}))$, F_s is obtained by adding/deleting one feature to the F_{in} and then each such feature is evaluated independently (stepwise selection).

A.3 Data transformation operations

These operations, modify the data set that can be given as inputs to the Explore and Evaluate operations of our framework. It can be denoted as $D^i = Q(D)$. We consider *SPJUA* queries Q for our data transformations.

[10] The shuffling of the dataset, which is required for sampling without replacement, is performed once a priori and the shuffled dataset is assumed to be used subsequently for the full program.

B Batching Optimization

We now present the extended problem statements, and algorithms for batching optimization, as mentioned in Section 3.2.1.

B.1 BatOptCPU: Batching with CPU costs

We now consider both scan costs and CPU costs of nodes when assigning nodes to batches. Direct-Memory-Access (DMA) will now impact which nodes get assigned to which batch. It will try to force each batch to become CPU-bound. We formulate the BatOptCPU problem as an ILP.

Entities:

- Given dataset $R(\mathcal{F})$ with full feature set $\mathcal{F}$
- DAG of scan-based nodes $G(V, E), |V| = N$. E imposes a dependency partial order $\leq$ on V, and the height of the partial order is H.
- Feature sets of the nodes, $f : V \to 2^{\mathcal{F}}$.
- A *batch* is a set of nodes, $b \subseteq V$ juxtaposed onto the same scan of R.
- Clearly, the minimum number of batches is H. The problem is to choose an allocation of nodes to batches that minimizes the overall cost.

Inputs:

- S, scan cost of $R(\mathcal{F})$.
- $c : V \to \mathbf{R}^+$, CPU costs of each node

Variables:

- $X : V \times \{1 \dots H\} \to \{0, 1\}$, indicators to assign a node to a batch

Objective: Minimize: Sum of each batch's MAX(Scan, total CPU)

$$\sum_{h=1}^{H} \max\{S, \ \sum_{v \in V} X(v, h)c(v)\}$$

Constraints:

- $\sum_{i=1}^{h} X(v, h) = 1, \ \forall v \in V$, i.e., a node is assigned to exactly 1 batch
- $X(u, h) + X(v, h) \leq 1, \ \forall (u, v) \in E, \ \forall h = 1 \dots H$, i.e., connected nodes cannot be in the same batch
- $\sum_{i=1}^{h} X(u, i) \geq \sum_{j=1}^{h} X(v, j), \ \forall (u, v) \in E, \ \forall h = 1 \dots H$, i.e., prevent reordering

Note that when $H = 1$ (i.e., $E = \phi$), the problem becomes trivial – just assign all nodes in V to a single batch. For $H > 1$ (i.e, $E \neq \phi$), it appears like a classical resource allocation problem.

Theorem B.1. *Given $G(V, E)$ with $E \neq \phi$, the scan cost S, and CPU costs of each node in V, BatOptCPU is* NP*-Hard in the width of the partial order imposed by E on V.*

Proof: We prove by reducing the NP-Hard problem of SetPartition to our problem. Given an instance of SetPartition, i.e. a multiset of positive integers $T = \{a_1, a_2, \ldots, a_N\}$, partition it into two subsets T_1 and T_2 such that $\sum_{x \in T_1} x = \sum_{y \in T_2} y = \frac{1}{2} \sum_{z \in T} z = w$ *(say)*. We now construct an instance of BatOptCPU as follows. $G(V, E)$ has $N + 2$ nodes, with two special nodes v_1, v_2 satisfying $v_1 < v_2$. Set the CPU cost of $c(v_1) = c(v_2) = b$, where $b \in \mathbf{Z}^+$ is some constant. The remaining N nodes have no dependencies, which means the width of the partial order in G is $N + 1$. These N nodes correspond to the elements of T – their CPU costs are set as the elements from T. Set the scan cost $S = b + w$. Clearly, the height of the partial order on G is 2, which means the optimal number of batches is 2. Wlog, let v_1 be in batch 1 (so v_2 is in batch 2).

We now claim that the given SetPartition instance has a solution if and only if the optimal cost of our problem instance is $2S$. Suppose there is a solution (T_1, T_2) to the SetPartition instance. Then, simply read the elements in T_1, and assign its corresponding nodes in G to batch 1. Repeat for 2. So the cumulative CPU cost in each batch is $w + b = S$, which means we now have the optimal solution cost of $2S$ for our problem instance. Conversely, if our problem instance has a solution with cost $2S$, then by the pigeonhole principle, it means that the CPU costs of the independent N nodes were1 split evenly across the two batches. Simply read the nodes in batch 1 (other than v_1) and assign its corresponding elements in T to T_1. Repeating for 2, we get a solution to the given SetPartition instance. $\square$

B.2 Batching under Memory Constraints

Our batching problem as formulated before packs the nodes of the DAG $G(V, E)$ into a chain of nodes $G'(V', E')$. We now consider a memory-constrained setting, where we cannot batch nodes indefinitely. More precisely, we are given a system memory budget M, and memory footprints of each node in G as $f : V \to \mathbf{R}^+$ (we assume $f(v) \leq M, \forall v \in V$). Our goal is now to pack the nodes of G into a sequence of batches such that we minimize the number of batches.

Theorem B.2. *Given $G(V, E)$, a memory limit M, and the memory footprints of all nodes in V, batching optimization is* NP*-Hard in the width of the partial order imposed by E on V.*

Proof: The problem can easily be reduced from BinPacking. Given a bin size B, objects to pack $\{x_i\}$ with sizes $\{s_i\}$, we construct an instance of our problem as follows: set $M = B$. Create $G(V, E)$ such that V corresponds exactly to $\{x_i\}$, and $E = \phi$. Set the memory footprints of the nodes in V as the the corresponding sizes in $\{s_i\}$ The problem instances are now equivalent, and thus an optimal solution to our instance will be an optimal solution to the given instance of BinPacking. $\square$

First-Fit Heuristic: Consider nodes from V that form an *antichain*, i.e., no pair among them is comparable with respect to the partial order imposed by E on V. There are no dependencies among the nodes in an antichain and so they can all go into the same batch. Note that the length of the longest such antichain is by definition the width of the partial order on G. Our heuristic algorithm is given in Algorithm 2. Since E imposes a partial order on V, our DAG $G(V, E)$ we will have at least as many antichains as the height of this partial order. We simply apply Algorithm 2 to each antichain individually. We construct the antichains in time $O(|V|+|E|)$ using a topological sort of $G(V, E)$. Since the antichains need not be unique, we use the following technique to obtain a unqiue set of antichains – We simply assign a node in V to an antichain k if k is the earliest possible occurrence of that node in any topological order on V (i.e., how “far” is that node from the “start” of the DAG).

Algorithm 2: First-Fit Heuristic used for Batching under Memory Constraints

```
Data: Antichain of nodes $W \subseteq V$, Memory footprints $f : W \to \mathbf{R}^+$, Memory limit $M$
Result: Number of batches $X$, and allocation $a : W \to \{0, 1, \ldots, X\}$
$X \leftarrow 1$
$a(w) \leftarrow 0, \forall w \in W$ //initialization
foreach $w \in W$ do
    for $x$ = 1 to $X$ do
        if $f(w) + \sum_{u \in W : a(u) = x} f(u) \leq M$ then
            $a(w) \leftarrow x$
            break
        end
    end
    //$w$ is still not alloted to any batch
    if $a(w) = 0$ then
        $X \leftarrow X + 1$
        $a(w) = X$
    end
end
```

C Materialization Optimization

We first restate Theorem 3.1 from the body, and give its proof.

Theorem C.1. *Our dynamic program solves MatOpt for chains in time $O(N^2)$, with space $O(N^2)$.*

Proof: With N input feature sets, the dynamic program fills the lower triangular part of the $N \times N$ memo (excluding diagonal elements) in time $\Theta(N(N-1)/2)$. Filling the diagonal elements requires referring to each row above (upto the diagonal element), is in time $\Theta(N(N-1)/2+N)$. Reading the backpointers is in time $\Theta(N)$. Thus, the overall algorithm is in time $O(N^2)$. The memo of time costs in in space $\Theta(N(N-1)/2+N)$, while the backpointers are in space $\Theta(N)$. Thus, the overall space complexity is also $O(N^2)$. $\square$

C.1 Materialization with Computation Costs

We now consider CPU costs in addition to scan and materialization costs for the formulation of MatOpt from Section 3.2. The CPU cost of an atomic operation *op* that accesses a feature set $F \subseteq \mathcal{F}$ is denoted $Compute_{op}(F)$. We assume that each feature set in $\mathcal{S} = \{F_1, \dots, F_N\}$ is associated with a unique CPU cost. Thus, we drop the subscript *op* in $Compute_op$ for notational convenience. This means we can have some feature sets $F_i, F_j \in \mathcal{S}$ such that $F_i = F_j$, but $Compute(F_i) \neq Compute(F_j)$. Note that the chain assumption still holds, i.e., $\mathcal{F} \supseteq F_1 \cdots \supseteq F_N$. We define $Access(x,y) = \max(Compute(x), Scan(y))$, where $x, y \in \mathcal{S} \cup \{\mathcal{F}\}$ and $x \subseteq y$. Our problem MatOpt now has a slight change.

$$\begin{aligned}
&\min_{S' \subseteq \mathcal{S}} \; Cost(S'), \\
&where\ Cost(S') = \sum_{s \in S'} \min_{t \in S' \cup \{\mathcal{F}\} \setminus \{s\}:\ s \subseteq t} Store(t,s) \\
&\qquad + \sum_{s \in S} r(s) \min_{t \in S' \cup \{\mathcal{F}\}:\ s \subseteq t} Access(s,t)
\end{aligned}$$

Dynamic Program The dynamic program for $T(i,j)$ is modified slightly as in Algorithm 3. Since only the access cost inputs (which are constants) are different here, the algorithm's complexity is unaffected.

Algorithm 3: Dynamic Program for MatOpt with chain input and computation cost.

```
    //$\forall i \geq 0,$ entries using full dataset
$T(i+1,0) = r(F_{i+1})\ Access(F_{i+1},\mathcal{F})\ +\ T(i,0)$

    //$\forall j \leq i,$ entries using last materialization
$T(i+1,j) = r(F_{i+1})\ Access(F_{i+1},F_j)\ +\ T(i,j)$

    //$\forall i \geq 1,$ diagonal entries with materialization costs
$T(i,i) = \min_{j=1\ldots i-1}\ (Store(F_i,F_j) + T(i-1,j))\ +\ r(F_i)\ Access(F_i,F_i)$
```

Algorithm 4: Dynamic Program for MatOpt on a k-width Lattice.

```
    //$i \geq 0:$
$T(i+1,\mathbf{0}) = T(i,\mathbf{0}) + \sum_{j=1\ldots k} r(F(i+1,j))Scan(\mathcal{F})$

    //$i \geq 0; \mathbf{v}$ s.t. $\forall j = 1\ldots k, v_j \leq i:$
$T(i+1,\mathbf{v}) = T(i,\mathbf{v}) + [\sum_{j=1\ldots k} r(F(i+1,j))]\ \min_{l=1\ldots k} Scan(F(v_l,l))$

    //$i \geq 1; \mathbf{v}$ s.t. $\forall j = 1\ldots k, v_j \leq i,$ and $\exists j = 1\ldots k,$ s.t. $v_j = i:$
$T(i,\mathbf{v}) = \sum_{j:v_j=i} (r(F(i,j))Scan(F(i,j)))+$
    $[\sum_{j:v_j<i} r(F(i,j))]\ \min_{l=1\ldots k} Scan(F(v_l,l))+$
    $\min_{\mathbf{w}:w_j<i,\forall j:v_j=i,\ and\ w_j=v_j,\forall j:v_j<i} [T(i-1,\mathbf{w}) + \sum_{j:v_j=i} \min_{l=1\ldots k} Store(F(i,j),F(w_l,l)]$
```

C.2 Materialization with Bounded-width Lattice

We now study MatOpt for the input feature sets existing in a bounded-width lattice. We will then prove Theorem 3.2 from Sec 3.2 in the body. Recall the setup: the feature sets in S have a lattice ordering $L : (S, \subseteq)$. Let the height of L be N (i.e., we have N *levels* $L_1 \dots L_N$). The width of L is a given constant k (i.e., k sets at each level). Sets in $L_1 \dots L_{N-1}$ have k children each, and sets in $L_2 \dots L_N$ have k parents each. In this case, our dynamic program's memo is richer and has k entries per slot, corresponding to the width of the lattice. In this scenario, we can still compute an optimum in time polynomial in N (but exponential in the constant k).

We introduce a convenient "grid" notation for the feature sets in S that we will use in the rest of this section. We write $S = \{F(i,j)\},\ i \in \{1,\dots,N\}\ \textit{and}\ j \in \{1,\dots,k\}$. A level is thus a set of feature sets with the same i, while we call a set of feature sets with the same j as a "column".

Dynamic Program: Our memo is changed to $T(i,\mathbf{v})$, which is the cumulative *Cost* upto level i, and $\mathbf{v} = (v_1,\dots,v_k)$ is a k-length vector of indices, each of which refers to the level at which the last materialization was done in its respective column ($0 \le v_j \le N, \forall j = 1\dots k$). We define $T(i,\mathbf{v})$ recursively (considering only scan and materialization costs) as given in Algorithm 4.

Theorem C.2. *Our dynamic program solves* MatOpt *for bounded-width lattices in time $O(k^2 2^k N^{k+1})$ with space $O(N^{k+1})$.*

Proof: The proof is along the lines of that for Theorem 3.1. In our memo $T(i,\mathbf{v})$, i can take N values, and $\mathbf{v}$ is a k-length vector in which each entry v_j can take one of N values. Thus the size of the memo is $O(N \times N^k)$. The backpointers require a space of only $\Theta(N)$, making the overall space complexity $O(N^{k+1})$.

Fixing i, there is 1 entry of the form $T(i,\mathbf{0})$, computing which is in time $\Theta(k)$. So we get total time $O(Nk)$ here.

Fixing i, there are $i^k - 1$ entries of the form $T(i,\mathbf{v}),\ s.t.\ \forall j, v_j \le i-1,\ \textit{and}\ \mathbf{v} \ne \mathbf{0}$. Computing each is in time $\Theta(k)$. So we get total time $k\sum_{i=1}^{N}(i^k - 1) = O(N^{k+1})$ here.

Finally, fixing i, the entries of the form $T(i,\mathbf{v}),\ s.t.\ \forall j, v_j \le i,\ \textit{and}\ \exists j,\ s.t.\ v_j = i$ and their total time can be computed using a z to count how many v_j exist such that $v_j = i$. So the time spent for a fixed i is:

$$\begin{aligned}
t(i) &= \sum_{z=1}^{k}\binom{k}{z} i^{k-z}(z + (k-z)k + i^z zk) \\
&= ki^k \sum_{z=1}^{k}\binom{k}{z} z + k^2 \sum_{z=1}^{k}\binom{k}{z} i^{k-z} - (k-1)\sum_{z=1}^{k}\binom{k}{z} i^{k-z} z \\
&\le ki^k(k2^{k-1}) + k^2 i^k[(1+i^{-1})^k - 1] \\
&\le k^2 2^k i^k + k^2[(i+1)^k - i^k]
\end{aligned}$$

So the total time we get here is $\le \sum_{i=1}^{N} t(i) \le k^2 2^k N^{k+1} + k^2(N+1)^k \le ck^2 2^k N^{k+1}$, for some constant c. Thus, the time here is $O(k^2 2^k N^{k+1})$. So the overall time is also $O(k^2 2^k N^{k+1})$. □

C.3 General set of sets

We now prove Theorem 3.3 from Section 3.2 of the body.

Theorem C.3. MatOpt *is* NP*-Hard in the width of the lattice on $\mathcal{S}$, the set of input feature sets.*

Proof: We prove by reduction from the NP-Hard problem of SetCover-EC We are given a universe of elements $\mathcal{F}$, and a family $\mathcal{S}$ of subsets of $\mathcal{F}$ that covers $\mathcal{F}$, i.e., $\cup_{s \in \mathcal{S}} s = \mathcal{F}$, such that $s \in \mathcal{S}$ has equal cardinality L. The optimization version of SetCover-EC asks for a sub family $S' \subseteq \mathcal{S}$ of minimum $|S'|$ that covers $\mathcal{F}$. We reduce an instance of SetCover-EC to an instance of our problem as follows:

$\mathcal{F}$ is the treated as the full set of features. Let $|\mathcal{S}| = N$. Create N feature set inputs $\{F_s | F_s \subseteq \mathcal{F}\}$, one for each $s \in \mathcal{S}$. We use $\mathcal{S}$ for $\{F_s\}$ as well. All F_s equal cardinality L. Create $|\mathcal{F}|$ additional feature singleton set inputs $\{x_j\}$, representing singleton feature sets from $\mathcal{F}$. We use X to denote $\{x_j\}$. We now construct other inputs to our problem conforming to our cost model:

Firstly, make CPU costs negligible so that data access (scan) times are linear only in the number of features in the dataset. It is given by $Scan(F_j) = \alpha|F_j|$, where α is a free parameter. Note that it does not depend on F_i, which is the input for computations.

Secondly, materialization costs are also linear, and given by $Store(F_i) = Scan(\mathcal{F}) + Write(F_i) = \alpha|\mathcal{F}| + \beta|F_i|$, where β is also a free parameter. It is reasonable to assume that only $\mathcal{F}$ is used for materialization since we can always make $\mathcal{S}$ an antichain by dropping a set contained in another within $\mathcal{S}$ in the given SetCover-EC instance.

Thirdly, set the repetitions of all F_s to be equal to γ, another free parameter. Set the repetitions of every x_j to 1.

The idea is that members of X will never be materialized since their individual materialization costs are higher than just accessing $\mathcal{F}$, or any member of $\mathcal{S}$. So, we set the parameters of our problem instance in such a way that our optimal solution will materialize a subset of $\mathcal{S}$ to use both for serving those feature sets covers all of X. This optimal materialized subset will be an optimum for the given SetCover-EC instance.

Recall that we have 3 free parameters – α, β, and γ. The given instance of SetCover-EC gives us $\mathcal{F}$ (the universe of features), $\mathcal{S}$ (the family of subsets of $\mathcal{F}$), N ($= |\mathcal{S}|$), and L (cardinality of each member of $\mathcal{S}$). Our problem's objective function ($S' \subseteq \mathcal{S}$)is given by:

$$
\begin{aligned}
Cost(S') = & \sum_{s\in S'}(Store(s) + \gamma Scan(F_s)) + \sum_{s\in\mathcal{S}\setminus S'} \gamma Scan(\mathcal{F}) \\
& + \sum_{x\in X:\exists s\in S', x\subseteq s} Scan(F_s) + \sum_{x\in X:\forall s\in S', x\not\subseteq s} Scan(\mathcal{F}) \\
& \\
& = |S'|((\alpha|\mathcal{F}| + \beta L) + \gamma\alpha L) + (|\mathcal{S}| - |S'|)\gamma\alpha|\mathcal{F}| \\
& + \alpha L(\#x_j \textit{ covered by } S') + \alpha|\mathcal{F}|(\#x_j \textit{ not covered by } S') \\
& \\
& = |\mathcal{S}|\gamma\alpha|\mathcal{F}| \ + \ |S'|(-\gamma\alpha|\mathcal{F}| + \alpha|\mathcal{F}| + \beta L + \gamma\alpha L) \\
& + \alpha L|X| \ + \ \alpha(|\mathcal{F}| - L)(\#x_j \textit{ not covered by } S')
\end{aligned}
$$

The terms $|\mathcal{S}|\gamma\alpha|\mathcal{F}|$ and $\alpha L|X|$ above are constants independent of S', and so we can drop them from the optimization. Thus, we rewrite as :

$$Cost(S') = |S'|(\alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L)) + \alpha(|\mathcal{F}| - L)(\#x_j \textit{ not covered by } S')$$

Now, $|\mathcal{F}|$, L, and $N(= |\mathcal{S}|)$ are given by the instance. We demonstrate a choice of α, β, γ such that the optimal solution to the above gives the optimal to the SetCover-EC instance:

Firstly, we want the coefficient of $|S'|$ above (call it p) to be positive, i.e., $p = \alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L) > 0$. This can be done by fixing any positive value for α. Then fix any positive integer value for γ. Then, we choose a positive β s.t. $\beta > \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L$. Call the RHS d (say), i.e., $d := \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L$, and $\beta > d$.

Secondly, we need to see if the maximum value of the first term (which is Np, when $S' = \mathcal{S}$) is outweighed by an occurrence of the second term (i.e., when at least one x_j is not covered). In other words, we check if $Np < \alpha(|\mathcal{F}| - L)$. Expanding p, it becomes if $N(\alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L)) < \alpha(|\mathcal{F}| - L)$. Rearranging in terms of β, it becomes if $\beta < \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L \ + \ \alpha(|\mathcal{F}| - L)/(LN)$, i.e., if $\beta < d \ + \ \alpha(|\mathcal{F}| - L)/(LN)$. Since the second term is positive (as $|\mathcal{F}| > L$), we only have to choose β such that $d < \beta < d \ + \ \alpha(|\mathcal{F}| - L)/(LN)$.

Since not covering even one x_j raises the cost more than choosing all the sets, a minimum cost solution to our problem instance will always cover all x_j, and will minimize among the chosen subsets of S. Thus, an optimal solution to our problem instance as constructed above will be an optimal solution to the given instance of SetCover-EC. □

C.4 Online optimization

We now make one simple relaxation to introduce the online setting – not all repetitions, $\{r(F_i)\}_{i=1}^{N}\}$, are known up front. The motivation is that the number of iterations for an operation (e.g., a coefficient learner) is not known up front due to its stopping conditions.

Notation and Problem Description We see a *sequence* (in time) of feature sets $F(t), t = 1, 2, ...$, where at iteration t, we have a data access with feature set input $F(t)$. Our assumption of chain structure among the feature sets is unchanged, i.e., $\mathcal{F} \supseteq F(t_1) \supseteq F(t_2)....$ The problem is to devise a strategy that decides online whether or not to materialize a feature set with the aim of improving overall performance. We solve it by adapting the standard "break-even" algorithm for the ski-rental problem. We count the *waste* $W(t)$ accurued across iterations, tracking the potential performance savings lost by not taking the optimal decision (which could have been done had we known the iterations a priori).The waste evolves with the iterations, and helps decide the materialization of F_i, as explained in Algorithm 5.

Algorithm 5: Online Algorithm for materialization optimization.

```
Initialize W(0) = 0
Before beginning some $t \geq 1$:
if $\pi_{F(t)}(R)$ is present then
    //no waste incurred here: use $\pi_{F(t)}(R)$
    $W(t) = W(t-1)$
else
    //$F^*$ is the last materialized dataset (could be $\mathcal{F}$ originally): use $\pi_{F^*}(R)$
    $W(t) = W(t-1) + Scan(F^*) - Scan(F(t))$
After some $t \geq 1$:
if $W(t) \leq Store(F(t+1))$ and $W(t) + Scan(F^*) - Scan(F(t+1)) > Store(F(t+1))$ then
    Materialize $\pi_{F(t+1)}(R)$, and replace $F^* \leftarrow F(t+1)$
```

Theorem C.4. *The Online Algorithm yields a solution whose cost is $\leq 2\times$ the cost of the optimal (i.e., optimization with repetitions known a priori).*

Proof: Consider the period from the start upto a t^* when the first feature set is materialized by the Online Algorithm. Thus, we have our waste:

$$W(t^*) = t^* Scan(\mathcal{F}) - \sum_{t=1}^{t^*} Scan(F(t) \tag{1}$$

$$W(t^*) \leq Store(F(t^*+1)), \text{ and} \tag{2}$$

$$W(t^*) + Scan(\mathcal{F}) - Scan(F(t^*+1)) > Store(F(t^*+1)) \tag{3}$$

Since we will materialize $F(t^*+1)$, our cost upto and including t^*+1 will be:

$$OnlineCost(t^*+1) = [\sum_{t=1}^{t^*} Scan(\mathcal{F})] + Store(F(t^*+1)) + Scan(F(t^*+1)) \tag{4}$$

Now, we claim that the Optimal must have materialized at least one feature set at or before $t^* + 1$. Using the claim, we provide the following lower bound on its cost:

$$OptimalCost(t^* + 1) \geq \sum_{t=1}^{t^*} Scan(F(t)) + Store(F(t^* + 1)) \tag{5}$$

Clearly, the first part of the lower bound is the cumulative scan cost, which is inevitable. The claim to prove is that the materialization cost of $F(t^* + 1)$ is admissible in the lower bound. We show that it is admissible because the Optimal must have materialized some feature set $F(t')$, where $t' \leq t^* + 1$. Recall that since $F(t') \supseteq F(t^* + 1)$, $Store(F(t^* + 1))$ is a lower bound on the materialization cost of any feature set that came before. We now prove the lower bound claim by contradiction. Assume the Optimal did not materialize anything till and including $t^* + 1$. So, it would have used $\mathcal{F}$ to serve all feature sets upto and including $t^* + 1$. But since Optimal yields a minimum cost, the following should hold:

$$(t^* + 1)Scan(\mathcal{F}) \leq \sum_{t=1}^{t^*+1} Scan(F(t)) + Store(F(t^* + 1)) \tag{6}$$

Substituting the waste from Inequality 1 into the above, we get:

$$W(t^*) + Scan(\mathcal{F}) - Scan(F(t^* + 1)) \leq Store(F(t^* + 1) \tag{7}$$

The above inequality contradicts Inequality 3. Hence, the lower bound in Inequality 5 holds. Now, we subtract Inequality 5 and Equation 4, and then use Inequalities 2 and 5.

$$\begin{aligned}
OnlineCost(t^* + 1) - OptimalCost(t^* + 1) \leq W(t^*) + Store(F(t^* + 1)) + \\
Scan(F(t^* + 1)) - Store(F(t^* + 1)) - Scan(F(t^* + 1)) \\
= W(t^*) \\
\leq Store(F(t^* + 1)) \\
\leq OptimalCost(t^* + 1)
\end{aligned}$$

Thus, we get the 2-approximation for this interval.

$$OnlineCost(t^* + 1) \leq 2 \times OptimalCost(t^* + 1) \tag{8}$$

We can repeat the same argument as above for the next internal, i.e., between the first and second materialized feature sets of the Online Algorithm. For the second interval, $F(t^*+1)$ will play the role of $\mathcal{F}$ above, and we will again get the 2-approximation. Since the 2-approximation holds in each interval, adding them all up means the Online Algorithm yields an overall 2-approximation to the Optimal. $\square$

D Provenance Management

For the explore and evaluate operations defined in our frame work, provenance information can be maintained at different level of granularities. Let us define an operation Op as $O = Op(\{I\})$, where I is the set of inputs and O is the output. The operation Op can be either *atomic* or work flow of atomic operations as in `EvaluateLRCV`. We now define *coarse* and *fine* grained level of workflow provenance.

Definition D.1. *Coarse grained : Let $W = Op_1 \circ Op_2 \circ ..Op_n$. Consider an instance of logical operation W as $(Op_1 \circ Op_2 \circ ..Op_n)(I_1, I_2, ..., I_n) = O$, with initial input $\{I_1, I_2, ..., I_n\}$. The coarse grained provenance of W, denoted $P_W(O)$, is given by $\{I_1, I_2, ..., I_n\}$.*

Definition D.2. *Fine grained : Let $W = Op_1 \circ Op_2 \circ ..Op_n$. Consider an instance of logical operation W as $(Op_1 \circ Op_2 \circ ..Op_n)(I_1, I_2, ..., I_n) = O$, with initial input $\{I_1, I_2, ..., I_n\}$. If e is the initial input element then $P_W(e) = \{e\}$.If e is any intermediate output, let $P_{Op}(e)$ denote the one level provenance of e with respect Op. Then $P_W(e) = \bigcup_{e' \in P_{Op}(e')} P_W(e')$*

D.1 Provenance for each Operation

In this section, we shall define coarse and fine grained provenance for the operations defined in our framework.

D.1.1 Atomic Operations

Feature Insert and Delete The *add* and *delete* are atomic operations, that enable the user to incoporate the domain knowledge. The operation is given by $F_{out} = Op(F_{in1}, F_{in2})$, where F_{in1}, F_{in2} denote the input feature sets and F_{out} denote the output feature set. The provenance is given by $P_{op}(F_{out}) = \{F_{in1}, F_{in2}\}$

Automatic selection algorithms Automatic selection algorithms are considered as *off-the shelf* algorithms where the internals of the algorithm is assumed to be unknown.

Lasso Let F_{in} denote the input feature set, $\{P\}$ denote the input parameters and R denote the data set relation. Then the operation is denoted as $F_{out} = Op(F_{in}, \{P\}, R)$. The provenance is given by $P_{Op}(F_{out}) = \{F_{in}, \{P\}, R\}$

Learners Let F_{in} denote the input feature set, $\{P\}$ denote the input parameters. S_{in} denote the input state and R denote the data set relation. Then the operation is denoted as $S_{out} = Op(F_{in}, \{P\}, S_{in}, R)$, where S_{out} represents the output state . The provenance is given by $P_{Op}(S_{out}) = \{F_{in}, \{P\}, S_{in}, R\}$.

Scorers Let S_{in} denote the input co-efficients, F_{in} denote the input feature set $\{P\}$ denote the input parameters and let R denote the input data set relation. Then the scoring operation is denoted as $S = Op(F_{in}, \{P\}, S_{in}, R)$. The provenance is given by $P_{Op}(S) = \{F_{in}, \{P\}, S_{in}, R\}$.

D.1.2 Columbus Operations

Operations in Columbus are workflows of atomic operations. Recall that `EvaluateLRCV` and `EvaluateAIC` are workflows of atomic learners and scorers and stepwise selection operations (`StepAdd` and `StepDrop`) are workflows of `EvaluateLRCV` or `EvaluateAIC`. The coarse grained workflow provenance of Columbus operations records the all of inputs and output of each operation, i.e., feature set input values, dataset identifiers, and parameters. For example, for `EvaluateAIC`, we will record the feature set input and output AIC score alone. The fine grained workflow provenance records the provenance for each of the atomic operation in the workflow of the Columbus operation. For example, for `EvaluateAIC`, we will record the coefficent values after each iterations as well. And for a `StepAdd` with AIC scores, we will record the AIC scores (and coefficents) of each feature set visited inside the workflow of that `StepAdd`.

Data transforming operations We consider *Select, Project, Join, Union*, and *Aggregation* transformations in our framework. All composite SPJ transformations T can be transformed in to a sequence of algebraic transformations. And the transformations are maintained logically in terms of *attribute mappings* and *filters* as menitioned in [34]. For atomic operations like *Shuffle, Sort* we record the input dataset and its input paramters.

D.2 Matching workflows

Recall that Explore and Evaluate operations are workflows of atomic operations. In our framework we identify equivalence among operations simply using a syntactic subexpression matching. Thus, an operations of type `CoefficientLR` can only match with another `CoefficientLR`. The purpose of identifying such subexpression matches is to reuse the results (captured as provenance) of prior computations on the dataset, thus improving performance. We describe a simple algorithm that identifies such subexpression matches in the absence of any data transforming operations in Algorithms 6 and 7. Two Columbus programs are represented as workflows W_1, W_2, which are DAGs of Columbus operations.

Algorithm 6: Subwork flow equivalence in the absence of data transform operators

```
Data: Workflows $W_1(V_1, E_1)$, $W_2(V_2, E_2)$
Result: Returns True if $W_2$ is contained in $W_1$
initialization: $C_1$ = head($W_1$), $C_2$ = head($W_2$);
if $type(C_1) = type(C_2)$ then
    return subFlowMatch($C_1$, $C_2$)
else
    return False;
```

The above algorithm has time complexity $O(NM)$, where N is the number of nodes in W_1 and M is the number of nodes in W_2. In our system, all provenance information for Columbus operations are stored as relational tables (or equivalent forms) in memory. To check if a given Columbus operation instance can reuse prior computations, a simple lookup of the operation (along with its inputs and parameters) is done. If the operation was computed earlier, the results are used and the computation is suppressed. Note that our workflows can have Data-Transform

Algorithm 7: SubFlowMatch

```
Data: Logical nodes $v_1$, $v_2$ from the work flows $W_1$ and $W_2$
Result: Returns True if dag with root $v_2$ is contained in dag with root $v_1$
initialization: $C_1 \longleftarrow$ children($v_1$), $C_2 \longleftarrow$ children($v_2$), res $\longleftarrow$ False ;
if $C_2 = \phi$ then
    return True
if $C_1 = \phi$ then
    return False
for each child $c_2$ in $C_2$ do
    doneFlag $\longleftarrow$ False
    res $\longleftarrow$ False
    for each child $c_1$ in $C_1$ and doneFlag =False do
        if input and types of $c_1$ and $c_2$ match then
            res $\longleftarrow$ SubFlowMatch($c_1$, $c_2$)
            if res = True then
                doneFlag $\longleftarrow$ True
    if res = False then
        return False
return res
```

operations that represent relational SPJUA queries. We currently use a syntactic subexpression match to detect equivalence among the relational queries as well.

E OptBatMat: Joint Optimization of Batching and Materialization

We now discuss an improvement to our BatMatOpt strategy, which performs batching, followed by materialization optimization. In some scenarios, we can get faster plans by considering batching and materialization optimizations jointly. This is because batching enforces unions of feature sets to make them larger, while materializations takes advantage of small feature sets. This sets up a tradeoff with competing properties of the two optimizations. Since the materialization optimization is already NP-Hard, jointly optimizing for batching and materialization is clearly at least NP-Hard. We formulate the joint optimization as an ILP. Note that our strategy of BatMatOpt in Section 5 mitigates the hardness of joint optimization by performing batching first, followed by materialization optimization.

Assumptions:

- The only difference with the previous formulation is that here, we do not assume that all nodes at the same iteration are always batched. Instead, we assign nodes to batches, and batches to datasets.
- Thus, we start with a DAG (actually a forest of chains) of nodes, each representing one scan.

Entities:

- Given dataset $R(F)$ with full feature set F
- DAG of dependencies among nodes $G(V, E), |V| = N$. Let T be the length of the longest path in G (i.e., maximum number of iterations among operations)
- Feature sets of the nodes, $f : V \to 2^F$.
- $C = \mathit{Distinct}\{f(v)|v \in V\}$ be the distinct input feature sets. The set of candidates for materialization are $D = \{\cup_{c \in C_S} c | C_S \subseteq C\}$, i.e., each element of D is the union of the elements from a subset of C. Note that $C \subseteq D \subseteq 2^F$. Let $D' = D \cup \{F\}$ to add in the full dataset.
- Let $Q \subseteq V \times D'$ be the set of pairs obtained using feature set containments among nodes and candidates, i.e., $Q = \{(v, d)|v \in V, d \in D', f(v) \subseteq d\}$.
- A *batch* is a set of nodes, $b \subseteq V$ juxtaposed onto the same scan of a $d \in D'$ – called a *batched data access*. Note that this requires $(v, d) \in Q, \forall v \in b$.

Inputs:

- $m : C \to \mathbf{R}^+$, projection materialization costs (assuming only $R(F)$ is used as input)
- $p : V \to \mathbf{R}^+$, computation costs of each node
- $s : D' \to \mathbf{R}^+$, data access costs (of projected datasets)

Variables:

- $I : D' \to \{0, 1\}$, indicators to choose which candidates to materialize.
- $B : D' \times \{1 \dots T\} \to \{0, 1\}$, indicators to choose batched data accesses; each dataset in D' could have T sequential accesses, labeled $1..T$.
- $X : Q \times \{1 \dots T\} \to \{0, 1\}$, indicators to assign feasible node-candidate pairs to a sequence number

Objective:

$$\min_{I,B,X} \sum_{d \in D} I(d)m(d) + \sum_{d \in D'} \sum_{t=1..T} B(d,t)\ MAX\{s(d), \sum_{(v,d) \in Q} X((v,d),t)\ p(v)\}$$
$$(i.e.,\ materializations\ +\ each\ batch's\ max(Scan, CPU))$$

Constraints:

- $\sum_{(v,d) \in Q} \sum_{t=1..T} X((v,d),t)\ B(d,t)\ I(d) = 1, \forall v \in V$, a node is assigned to exactly 1 batched data access on a dataset that is present/materialized.
- $\sum_{(i,d) \in Q} X((i,d),t) + \sum_{(j,d) \in Q} X((j,d),t) \leq 1,$
  $\forall (i,j) \in E,\ \forall t = 1..T,$
  nodes connected in $G(V,E)$ cannot be assigned the sequence number.
- $\sum_{(i,d) \in Q} \sum_{t=1..T'} X((i,d),t) \geq$
  $\sum_{(j,d) \in Q} \sum_{t=1..T'} X((j,d),t),$
  $\forall (i,j) \in E,\ \forall T' = 1..T$, assigned sequence numbers have to preserve the ordering of nodes in G (to prevent "deadlocks").
- $I(F) = 1$, full dataset is always present.

Notes:

- The problem is a QP, since the objective has a product of variables (the MAX can be replaced with a variable, and more constraints).
- The first set of constraints are cubic, but they can be replaced with five sets of linear constraints:

  $\forall v \in V,\ \sum_{(v,d) \in Q} \sum_{t=1..T} X((v,d),t) = 1$
  i.e., each node goes to only 1 batched data acccess.

  $\forall (d,t) \in D' \times \{1..T\},$
  $\sum_{(v,d) \in Q} X((v,d),t) \leq M1 * B(d,t),\ and$
  $\sum_{(v,d) \in Q} X((v,d),t) - 1 \geq M2 * (B(d,t) - 1)$
  i.e. each batched data access exists if and only if at least one node uses it.

  $\forall d \in D,$
  $\sum_{t=1..T} B(d,t) \leq M3 * I(d)$
  $\sum_{t=1..T} B(d,t) - 1 \geq M4 * (I(d) - 1),$
  i.e. each dataset is materialized if and only if it has at least one batched data access.

  where M1..M4 are large ($> |Q|T$) positive constants.

F Experiments

F.1 Columbus Programs

F.1.1 Ford1

For the simplicity of exposition, we shall refer each feature by its index and the index starts from 0 (and goes up to 29 on the Ford dataset). The number of training iterations for `CoefficientLR`, `EvaluateLRCV`, `StepAdd`, and `StepDrop` is by default 5, unless otherwise indicated.

```
1. d1 = Dataset(c("Ford")) #Register the dataset
2. s1 = FeatureSet(c(0:9))
3. l1 = CorrelationX(s1, d1)
4. s2 = FeatureSet(c(1, 3, 5, 6))
5. l2 = EvaluateLRCV(s2, d1)
6. s3 = FeatureSet(c(4))
7. s4 = Union(s2, s3)
8. l4 = EvaluateLRCV(s4, d1)
9. s5 = FeatureSet(c(10:29))
10. s6 = StepAdd(s4, s5, d1, "AIC") #AIC score; exclude s5
11. s7 = FeatureSet(c(3, 4, 5, 6))
12. l5 = EvaluateLRCV(s7, d1)
13. s8 = FeatureSet(c(10:19))
14. l8 = CorrelationX(s8, d1)
15. s9 = FeatureSet(c(11, 14, 15 , 16))
16. l9 = EvaluateLRCV(s9, d1)
17. s10 = FeatureSet(c(20))
18. s11 = Union(s9, s10)
19. s12 = FeatureSet(c(19:29))
20. s13 = Union(s7, s9)
21. l10 = CoefficientLR(s13, d1) //#default parameters
22. l11 = CorrelationX(s12, d1)
23. s14 = FeatureSet(c(27, 28))
24. s15 = Union(s11, s14)
25. l12 = CoefficientLR(s15, d1)
26. l13 = EvaluateLRCV(s15, d1)
27. s16 = FeatureSet(c(11, 14, 15, 16, 28))
28. s17 = StepDrop(s16, d1, "AIC", 20) #AIC score; 20 iterations
```

Figure 10: Program Ford1.

F.1.2 Ford2

For the simplicity of exposition, we shall refer each feature by its index and the index starts from 0 (and goes up to 29 on the Ford dataset). The number of training iterations for `CoefficientLR`, `EvaluateLRCV`, `StepAdd`, and `StepDrop` is by default 50, unless otherwise indicated.

```
1. d1 = Dataset(c(“Ford”)) #Register the dataset
2. s1 = FeatureSet(c(0:4), d1))
3. s2 = FeatureSet(c(9:29), d1))
4. s3 = Union(s1,s2), d1))
5. l1 = CoefficientLR(s3, d1)
6. s4 = BestK(l1, 6) #Top 6 features
7. s5 = StepDel(s4, d1, “AIC”) #AIC score
8. s6 = StepDel(s5, d1, “AIC”) #AIC score
9. s7 = StepDel(s6, d1, “AIC”) #AIC score
10. s8 = FeatureSet(c(5:8), d1))
11. s9 = Union(s7,s8)
12. l2 = CoefficientLR(s9, d1)
13. s10 = BestK(l2, 5) #Top 5 features
14. l3 = CorrelationX(s10, d1)
15. s11 = FeatureSet(c(8:29))
16. s12 = StepAdd(s10, s11, d1, “AIC”)#AIC score; exclude s11
```

Figure 11: Program Ford2.

F.1.3 Census1

For the simplicity of exposition, we shall refer each feature by its index and the index starts from 0 (and goes up to 160 on the Census dataset). The number of training iterations for `CoefficientLR`, `EvaluateLRCV`, `StepAdd`, and `StepDrop` is by default 5, unless otherwise indicated.

```
1. d1 = Dataset(c("Census")) #Register the dataset
2. s1 = FeatureSet(c(6:13))
3. l1 = CorrelationX(s1, d1)
4. s2 = FeatureSet(c(6,8,10,12))
5. l2 = EvaluateLRCV(s2, d1)
6. s3 = FeatureSet(c(15:82))
7. s3p = FeatureSet(c(14))
8. l3 = CorrelationL(s3p,s3)
9. s4 = FeatureSet(c(15:24))
10. l4 = EvaluateLRCV(s4, d1)
11. s5 = FeatureSet(c(44:58))
12. l5 = EvaluateLRCV(s5, d1)
13. s6 = FeatureSet(c(84:128))
14. s6p = FeatureSet(c(129))
15. l6 = CorrelationL(s6p,s6)
16. s7 = FeatureSet(c(84:93))
17. l7 = EvaluateLRCV(s7, d1)
18. s8 = FeatureSet(c(131-161))
19. s8p = FeatureSet(c(130)
20. l8 = CorrelationL(s8p,s8)
21. s9 = FeatureSet(c(131:140))
22. l9 = EvaluateLRCV(s9, d1)
23. s10 = Union(s2,s4)
24. s11 = Union(s10,s7)
25. s12 = Union(s11,s9)
26. l0 = CoefficientLR(s12, d1)
27. s13 = BestK(l10, 11) #Top 11 features
28. s14 = StepDrop(s13, d1, "AIC", 20) #AIC score; 20 iterations
```

Figure 12: Program Census1.

F.1.4 Census2

For the simplicity of exposition, we shall refer each feature by its index and the index starts from 0 (and goes up to 160 on the Census dataset). The number of training iterations for `CoefficientLR`, `EvaluateLRCV`, `StepAdd`, and `StepDrop` is by default 50, unless otherwise indicated.

```
1. d1 = Dataset(c("Census")) #Register the dataset
2. s1 = FeatureSet(c(0:9), d1))
3. s2 = FeatureSet(c(13:160), d1))
4. s3 = Union(s1,s2), d1))
5. l1 = CoefficientLR(s3, d1)
6. s4 = BestK(l1, 20) #Top 20 features
7. s5 = StepDel(s4, d1, "AIC") #AIC score
8. s6 = StepDel(s5, d1, "AIC") #AIC score
9. s7 = StepDel(s6, d1, "AIC") #AIC score
10. s8 = FeatureSet(c(10:12), d1))
11. s9 = Union(s7,s8)
12. l2 = CoefficientLR(s9, d1)
13. s10 = BestK(l2, 10) #Top 10 features
14. l3 = CorrelationX(s10, d1)
15. s11 = FeatureSet(c(15:161))
16. s12 = StepAdd(s10, s11, d1, "AIC")#AIC score; exclude s11
```

Figure 13: Program Census2.

F.2 Batch Setting Scalability: Other Programs

Figure 14 plots the runtimes of the full programs Ford2 and Cens1 on their respective datasets. As with Cens2 in Figure 6 of Section 5, materialization in addition to batching yields more significant speedups than batching alone on Ford2. And similar to Ford1 in Figure 6 of Section 5, batching alone yields significant speedups on Cens1. However, materialization optimization increases the speedups significantly here because subsequent operations in Cens1 access less than half the features, even after batching.

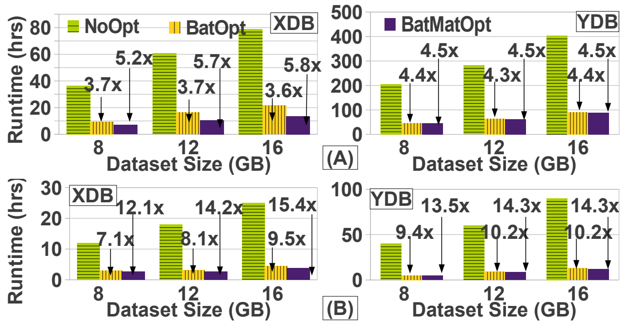


Figure 14: Batch setting for (A) Ford2 and (B) Cens1 on both systems as the dataset size is scaled up. NoOpt is naive execution, BatOpt applies batching, and BatMatOpt applies both batching and materialization optimizations. The speedups achieved by BatOpt and BatMatOpt against NoOpt are also shown.

F.3 Batching Optimization: StepAdd

Figures 15 and 16 plot the runtimes for `StepAdd` with and without batching optimization (within the operation across feature sets). Since the dataset has 30 features, an input set size of 5 means that the operations computes coefficients for 25 feature sets, each of size 6. Similarly, an input set size of 25 means we have 5 feature sets, each of size 26. Batching speeds up `StepAdd` by 2x-9x on SynFord-8GB, and by 5x-19x on SynFord-16GB.

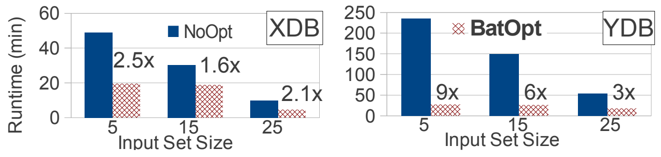


Figure 15: Runtimes of `StepAdd` (one training iteration, AIC). The dataset is SynFord-8GB.

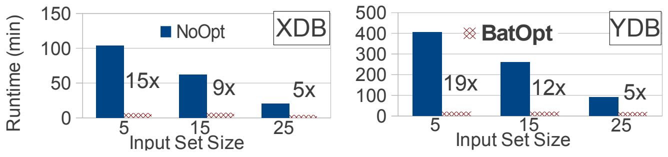


Figure 16: Runtimes of `StepAdd` (one training iteration, AIC). The dataset is SynFord-16GB.

F.4 Cost Model Validation

We now validate that our cost model is able to help our optimizer make accurate predictions of runtimes to choose optimal plans. Fixing a dataset $R(\mathcal{F})$, our cost model to capture the costs of projection materializations, scans, and computations is as follows:

- The cost of a scan on a dataset with features F, i.e., scanning $\pi_F(R)$, is denoted $Read(F)$ and modeled as a linear function $a + b|F|$. More precisely, to account for buffer memory residency of datasets, the scan cost is a two-piece linear function.
- The cost of materialization of F, i.e., writing $\pi_F(R)$ is denoted $Store(F)$ and also modeled as a linear function $c + d|F|$. For simplicity, we use the fixed full dataset $R(\mathcal{F})$ to perform materializations, although we can also model it as a function of two variables (denoting the number of features in the dataset and the number of features to project).
- The computation costs of individual operations are modeled as linear or quadratic functions of the input feature set size. Specifically, `CoefficientLR` and `EvaluateLRCV` are linear while `CorrelationX` is quadratic. `StepAdd` and `StepDrop` are modeled as a function of two variables – the number of feature sets they visit and the length of each feature set.

We obtain the values of the constants in the above cost formulae by performing a few offline passes over $R(\mathcal{F})$. Half a dozen values of F are chosen to obtain representatives value of the costs. We then perform linear (or quadratic, if needed) regression to obtain the final cost model values.

F.4.1 Cost Model: End-to-end

The optimizer uses the cost model to predict runtimes of alternate execution strategies. To verify the end-to-end accuracy of our cost model, we present the predicted and actual runtimes of a full program (Ford1) on two large synthetic datasets in Table 9.

Program on Dataset	**Strategy**	**Predicted**	**Actual**	**Error**
Ford1 on SynFord-12GB	NoOpt	71140	72691	2.1%
	BatOpt	11538	11217	2.8%
	BatMatOpt	8981	8941	0.4%
Ford1 on SynFord-16GB	NoOpt	94198	104258	9.6%
	BatOpt	15284	14717	3.8%
	BatMatOpt	12197	11879	2.7%

Table 9: Predicted and actual runtimes on a full program Ford1 on two large synthetic datasets for all three strategies. The costs of scans, materializations, and computations are all considered by the optimizer when predicting the runtimes. The runtimes are in seconds.

F.4.2 Cost Model: Materialization Optimization

We now validate that our cost model enables our optimizer to pick optimal materialization decisions. Recall Figure 8(A) from Section 5 that compared our optimizer's plan against two naive strategies – not materializing anything (NoMat) and materializing all projections (AllMat). The input here is a chain of three feature sets of lengths 5, 10, and 15 (out of 30 features) for I/O-bound operations (`CoefficientLR`). We fix all three repetitions to be equal, and vary the repetitions. We now present the runtimes of 8 possible plans, and compare how accurate our optimizer's predictions are. The 8 plans are numbered as follows: Plan 1 is NoMat. Plan 2 is to materialize $\{F_1\}$. Similarly, Plan 3: $\{F_2\}$, Plan 4: $\{F_3\}$, Plan 5: $\{F_1, F_2\}$, Plan 6: $\{F_2, F_3\}$, Plan 7: $\{F_1, F_3\}$, and Plan 8: AllMat. Figure 17 shows that various values of repetitions, our optimizer's predicted lowest-cost plan matches that actual fastest plan.

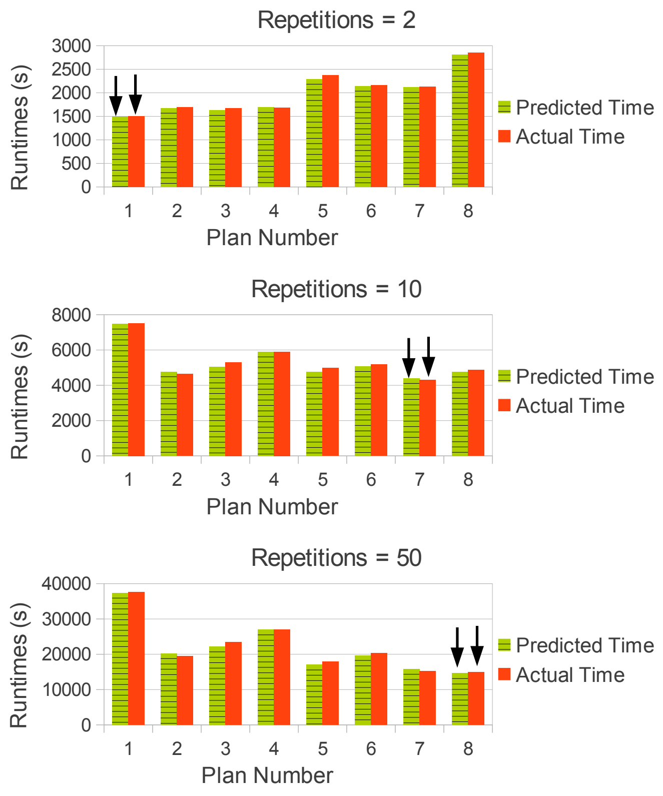


Figure 17: Predicted and actual runtimes across the space of 8 plans for the experiments in Figure 8(A) of Section 5 for 3 values of repetitions. The chosen lowest-cost plans (both predicted and actual) are indicated by arrows. The predicted optimal plan matches the actual fastest plan in all the three cases.

G Acknowledgements

We thank Xixuan Feng, Seeun Umboh, and Tyson Williams for proofreading an earlier version of the technical report.