

© 2019 Doris Xin

ACCELERATING HUMAN-IN-THE-LOOP MACHINE LEARNING

BY

DORIS XIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Assistant Professor Aditya Parameswaran

ABSTRACT

Machine learning workflow development is a process of trial-and-error: developers *iterate* on workflows by testing out small modifications until the desired accuracy is achieved. Unfortunately, existing machine learning systems focus narrowly on model training—a small fraction of the overall development time—and neglect to address iterative development. We propose HELIX, a machine learning system that *optimizes the execution across iterations*—intelligently caching and reusing, or recomputing intermediates as appropriate. HELIX captures a wide variety of application needs within its Scala DSL, with succinct syntax defining unified processes for data preprocessing, model specification, and learning. We demonstrate that the reuse problem can be cast as a MAX-FLOW problem, while the caching problem is NP-HARD. We develop effective lightweight heuristics for the latter. Empirical evaluation shows that HELIX is not only able to handle a wide variety of use cases in one unified workflow but also much faster, providing run time reductions of up to **19**× over state-of-the-art systems, such as DeepDive or KeystoneML, on four real-world applications in natural language processing, computer vision, social and natural sciences.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	BACKGROUND AND OVERVIEW	4
2.1	A Brief Overview of Workflows	4
2.2	Common Practices in Iteration	5
2.3	System Architecture	6
2.4	The Workflow Lifecycle	8
2.5	Example Workflow	9
CHAPTER 3	PROGRAMMING INTERFACE	11
3.1	Operations in ML Workflows	11
3.2	HML	15
3.3	Scope and Limitations	20
CHAPTER 4	COMPILATION AND REPRESENTATION	21
4.1	The Workflow DAG	21
4.2	Tracking Changes	21
CHAPTER 5	OPTIMIZATION	23
5.1	Preliminaries	23
5.2	Optimal Execution Plan	24
5.3	Optimal Materialization Plan	29
5.4	Workflow DAG Pruning	34
CHAPTER 6	EMPIRICAL EVALUATION	36
6.1	Systems and Baselines for Comparison	36
6.2	Workflows	37
6.3	Running Experiments	39
6.4	Metrics	40
6.5	Evaluation vs. State-of-the-Art Systems	40
6.6	Evaluation vs. Simpler HELIX Versions	43
6.7	Memory Usage by HELIX	44
CHAPTER 7	RELATED WORK	50
CHAPTER 8	CONCLUSIONS AND FUTURE WORK	54
REFERENCES	55

CHAPTER 1: INTRODUCTION

From emergent applications like precision medicine, voice-controlled devices, and driverless cars, to well-established ones like product recommendations and credit card fraud detection, machine learning continues to be the key driver of innovations that are transforming our everyday lives. At the same time, developing machine learning applications is time-consuming and cumbersome. To this end, a number of efforts attempt to make machine learning more declarative and to speed up the model training process [1].

However, the majority of the development time is in fact spent *iterating on the machine learning workflow* by incrementally modifying steps within, including (i) *preprocessing*: altering data cleaning or extraction, or engineering features; (ii) *model training*: tweaking hyperparameters, or changing the objective or learning algorithm; and (iii) *postprocessing*: evaluating with new data, or generating additional statistics or visualizations. These iterations are necessitated by the difficulties in predicting the performance of a workflow *a priori*, due to both the variability of data and the complexity and unpredictability of machine learning. ***Thus, developers must resort to iterative modifications of the workflow via “trial-and-error” to improve performance.*** A recent survey reports that less than 15% of development time is actually spent on model training [2], with the bulk of the time spent iterating on the machine learning workflow.

Example 1.1 (Gene Function Prediction) *Consider the following example from our bioinformatics collaborators who form part of a genomics center at the University of Illinois [3]. Their goal is to discover novel relationships between genes and diseases by mining scientific literature. To do so, they process published papers to extract entity—gene and disease—mentions, compute embeddings using an approach like word2vec [4], and finally cluster the embeddings to find related entities. They repeatedly iterate on this workflow to improve the quality of the relationships discovered as assessed by collaborating clinicians. For example, they may (i) expand or shrink the literature corpus, (ii) add in external sources such as gene databases to refine how entities are identified, and (iii) try different NLP libraries for tokenization and entity recognition. They may also (iv) change the algorithm used for computing word embedding vectors, e.g., from word2vec to LINE [5], or (v) tweak the number of clusters to control the granularity of the clustering. Every single change that they make necessitates waiting for the entire workflow to rerun from scratch—often multiple hours on a large server for each single change, even though the change may be quite small.*

As this example illustrates, the key bottleneck in applying machine learning is ***iteration***—every change to the workflow requires hours of recomputation from scratch, even though the

change may only impact a small portion of the workflow. For instance, normalizing a feature, or changing the regularization would not impact the portions of the workflow that do not depend on it—and yet the current approach is to simply rerun from scratch.

One approach to address the expensive recomputation issue is for developers to explicitly materialize all intermediates that do not change across iterations, but this requires writing code to handle materialization and to reuse materialized results by identifying changes between iterations. Even if this were a viable option, materialization of all intermediates is extremely wasteful, and figuring out the optimal reuse of materialized results is not straightforward. Due to the cumbersome and inefficient nature of this approach, developers often opt to rerun the entire workflow from scratch.

Unfortunately, existing machine learning systems do not optimize for rapid iteration. For example, KeystoneML [6], which allows developers to specify workflows at a high-level abstraction, only optimizes the one-shot execution of workflows by applying techniques such as common subexpression elimination and intermediate result caching. On the other extreme, DeepDive [7], targeted at knowledge-base construction, materializes the results of all of the feature extraction and engineering steps, while also applying approximate inference to speed up model training. Although this naïve materialization approach does lead to reuse in iterative executions, it is wasteful and time-consuming.

We present HELIX, a *declarative, general-purpose machine learning system that optimizes across iterations*. HELIX is able to match or exceed the performance of KeystoneML and DeepDive on one-shot execution, while providing ***gains of up to 19×*** on iterative execution across four real-world applications. By optimizing across iterations, HELIX allows data scientists to avoid wasting time running the workflow from scratch every time they make a change and instead run their workflows in time proportional to the complexity of the change made. HELIX is able to thereby substantially increase developer productivity while simultaneously lowering resource consumption.

Developing HELIX involves two types of challenges—challenges in *iterative execution optimization* and challenges in *specification and generalization*.

Challenges in Iterative Execution Optimization. A machine learning workflow can be represented as a directed acyclic graph, where each node corresponds to a collection of data—the original data items, such as documents or images, the transformed data items, such as sentences or words, the extracted features, or the final outcomes. This graph, for practical workflows, can be quite large and complex. One simple approach to enable iterative execution optimization (adopted by DeepDive) is to materialize every single node, such that the next time the workflow is run, we can simply check if the result can be reused from the

previous iteration, and if so, reuse it. Unfortunately, this approach is not only wasteful in storage but also potentially very time-consuming due to materialization overhead. Moreover, in a subsequent iteration, it may be cheaper to recompute an intermediate result, as opposed to reading it from disk.

A better approach is to determine whether a node is worth materializing by considering both the time taken for computing a node and the time taken for computing its ancestors. Then, during subsequent iterations, we can determine whether to read the result for a node from persistent storage (if materialized), which could lead to large portions of the graph being pruned, or to compute it from scratch. In this paper, we prove that the reuse plan problem is in PTIME via a non-trivial *reduction to* MAX-FLOW *using the* PROJECT SELECTION PROBLEM [8], *while the materialization problem is, in fact, NP-HARD.*

Challenges in Specification and Generalization. To enable iterative execution optimization, we need to support the specification of the end-to-end machine learning workflow in a high-level language. This is challenging because data preprocessing can vary greatly across applications, often requiring ad hoc code involving complex composition of declarative statements and UDFs [9], making it hard to automatically analyze the workflow to apply holistic iterative execution optimization.

We adopt a hybrid approach within HELIX: developers specify their workflow in an *intuitive, high-level domain-specific language (DSL) in Scala* (similar to existing systems like KeystoneML), using *imperative code as needed for UDFs*, say for feature engineering. This interoperability allows developers to seamlessly integrate existing JVM machine learning libraries [10, 11]. Moreover, HELIX is built on top of Spark, allowing data scientists to leverage Spark’s parallel processing capabilities. We have developed a GUI on top of the HELIX DSL to further facilitate development [12].

HELIX’s DSL not only enables automatic identification of data dependencies and data flow, but also encapsulates all typical machine learning workflow designs. Unlike DeepDive [7], HELIX is not restricted to regression or factor graphs, allowing data scientists to use the most suitable model for their tasks. All of the functions in Scikit-learn’s (a popular ML toolkit) can be mapped to functions in the DSL [13], allowing HELIX to easily capture applications ranging from natural language processing, to knowledge extraction, to computer vision. Moreover, by studying the variation in the dataflow graph across iterations, HELIX is able to identify reuse opportunities across iterations. Our work is a first step in a broader agenda to improve human-in-the-loop ML [14].

CHAPTER 2: BACKGROUND AND OVERVIEW

In this section, we provide a brief overview of machine learning workflows, describe the HELIX system architecture and present a sample workflow in HELIX that will serve as a running example.

A machine learning (ML) workflow accomplishes a specific ML task, ranging from simple ones like classification or clustering, to complex ones like entity resolution or image captioning. Within HELIX, we decompose ML workflows into three components: data preprocessing (DPR), where raw data is transformed into ML-compatible representations, learning/inference (L/I), where ML models are trained and used to perform inference on new data, and postprocessing (PPR), where learned models and inference results are processed to obtain summary metrics, create dashboards, and power applications. We discuss specific operations in each of these components in Chapter 3. As we will demonstrate, these three components are generic and sufficient for describing a wide variety of supervised, semi-supervised, and unsupervised settings.

2.1 A BRIEF OVERVIEW OF WORKFLOWS

A machine learning (ML) workflow accomplishes a specific ML task, ranging from simple ones like classification or clustering, to complex ones like entity resolution or image captioning. The more complex tasks are often broken down into smaller subtasks; e.g., image captioning is broken down into identifying objects or actions via classification, followed by generating sentences using a language model [15]. Within HELIX, we decompose ML workflows into three components: data preprocessing (DPR), learning/inference (L/I), and postprocessing (PPR). These three components are generic and adapt to a wide variety of supervised, semi-supervised, and unsupervised settings, as we will demonstrate in Section ??.

Let \mathcal{R} be the raw input data for the ML workflow.

Data Pre-processing (DPR). During *data preprocessing*, \mathcal{R} is transformed, through a series of operations, into some representation $\mathcal{D} \in \mathcal{X}$, where \mathcal{X} is the input for model training. The transformation from \mathcal{R} to \mathcal{D} can involve a variety of operations, such as fine-grained feature definition from individual attributes (e.g., number of vowels in a word), joining in other data sources (e.g., user information into log data), parsing (e.g., a document to words), and aggregation (e.g., aggregating ad clicks).

Learning/Inference (L/I). During *learning*, an algorithm is run on \mathcal{D} to obtain a model $f : \mathcal{X} \rightarrow \mathcal{Y}$, such as a linear classifier, decision tree, or cluster centers, where \mathcal{Y} denotes

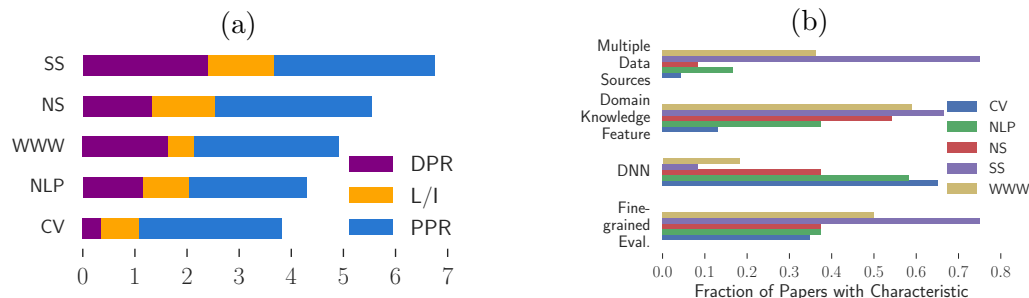


Figure 2.1: (a) The average DPR, L/I, and PPR iterations for each domain; (b) Fraction of papers with each characteristic by domain.

the space of the target outputs for the ML task. Then, during *inference*¹, this model f is used to process new data from \mathcal{X} . For example, in spam classification, f decides whether a new email is spam; in clustering, f assigns a datapoint to a specific (set of) cluster(s); in image captioning, f generates a text caption for an image; and in word embeddings, f maps a string containing a word onto a vector. We treat learning and inference as a unified component because data processed by the DPR component can either be used for learning, or, in the case a model has already been learned, inference. As shown in the examples above, this observation is valid for both supervised and unsupervised learning.

Post-processing (PPR). Finally, an ML workflow usually contains additional operations on the output model or inference results. This could include model evaluation, visualizations, or other application-specific activities. We refer to these operations as *postprocessing*.

2.2 COMMON PRACTICES IN ITERATION

ML workflow development is anecdotally regarded to be highly iterative [16, 17]. However, there is limited quantitative evidence to characterize iteration in ML workflow development. To this end, we analyzed a sample of applied ML papers. More details can be found in Xin et al. [18]; we provide a brief summary here. We collected statistics from papers from five application domains: computer vision (CV), natural language processing (NLP), web applications (WWW), natural sciences (NS), and social sciences (SS). The statistics collected pertain to the frequency of operations in the three workflow components introduced above. While academic papers present only a partial view of the development process, our survey nevertheless sheds light on common practices for iteration.

Each iteration is categorized by the workflow components modified in that iteration, e.g., an iteration in which the user changes learning parameters is an L/I iteration. Figure 2.1(a)

¹We use the term as defined in the ML community; this term is not to be confused with *statistical inference*, which is concerned with estimating distributions based on data.

shows the average number of DRP, L/I and PPR iterations for each application domain. Overall, the average (max) numbers of DPR, L/I, and PProc iterations are 1.3 (4), 1.2 (5), and 2.6 (5), respectively. The global average number of iterations is 5.1 across all domains; SS and NS exceed the global average; WWW is on par with the global average; CV and NLP fall below the global average. CV workflows have fewer DPR iterations, due to the fact that all surveyed CV papers study the same preprocessed datasets, and often use deep neural networks (DNNs). We found that by convention, CV and NLP papers report only the final model parameters and not the entire model tuning process, hence the below-average number of L/I iterations. NS papers tend to report on a larger number of models (e.g., SVM, Random Forest) since the applicability of a model class to the problem investigated presents value to future researchers (SVM is the most popular choice by a large margin).

In addition, we highlight four interesting characteristics discovered during our survey in Figure 2.1(b). First, WWW and SS are much more likely to incorporate *multiple data sources* in creating an ML model. Here, we focus on cases where a single model relies on multiple data sources, not the case where models are evaluated on multiple datasets. For WWW, this often entails joining log data with user profiles; SS often considers both the social network and auxiliary information such as geographic features. Second, except in CV, most domains still rely upon features handcrafted by domain experts. Third, contrary to common belief, DNNs are not ubiquitous, especially in SS and WWW, due in part to limited data and computing resources availability, and to the lack of human interpretability of outputs. Lastly, in addition to reporting aggregate metrics, authors often conduct fine-grained case studies on specific datapoints to study the limitations of their approach. Almost every CV paper contains case studies since images lend themselves to visualization. Furthermore, NS papers often study specific high-impact features to derive new scientific insights.

We use iteration trends and insights presented in Figure 2.1 to guide our system design and empirical evaluation.

2.3 SYSTEM ARCHITECTURE

The HELIX system consists of a domain specific language (DSL) in Scala as the programming interface, a compiler for the DSL, and an execution engine, as shown in Figure 2.2. The three components work collectively to *minimize the execution time for both the current iteration and subsequent iterations*:

- 1. Programming Interface.** HELIX provides a single Scala interface named *Workflow* for programming the entire workflow; the HELIX DSL also enables embedding of imperative

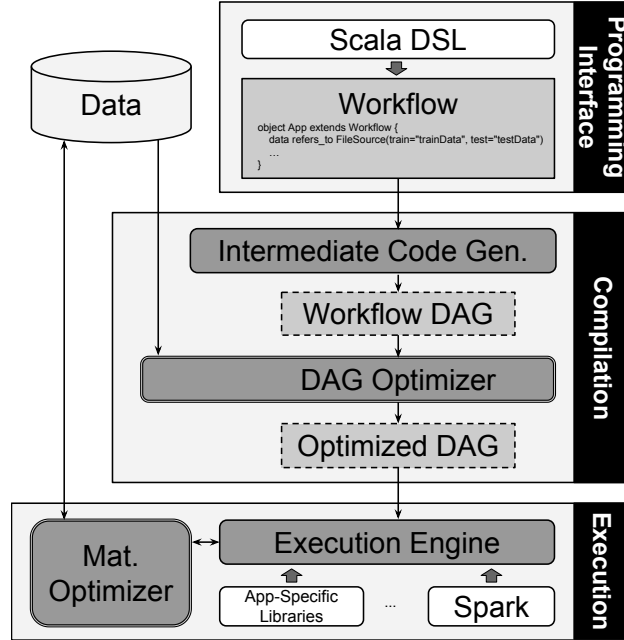


Figure 2.2: HELIX System architecture. A program written by the user in the HELIX DSL, known as a *Workflow*, is first compiled into an intermediate DAG representation, which is optimized to produce a physical plan to be run by the execution engine. At runtime, the execution engine selectively materializes intermediate results to disk.

code in declarative statements. Through just a handful of extensible operator types, the DSL supports a wide range of use cases for both data preprocessing and machine learning.

2. Compilation. A *Workflow* is internally represented as a directed acyclic graph (DAG) of operator outputs. The DAG is compared to the one in previous iterations to determine reusability (Chapter 4). The *DAG Optimizer* uses this information to produce an optimal *physical execution plan* that *minimizes the one-shot runtime of the workflow*, by selectively loading previous results via a MAX-FLOW-based algorithm (Section 5.1–5.2).

3. Execution Engine. The execution engine carries out the physical plan produced during the compilation phase, while communicating with the *materialization operator* to *materialize* intermediate results, to *minimize runtime of future executions*. The execution engine uses Spark [19] for data processing and domain-specific libraries such as CoreNLP [20] and Deeplearning4j [21] for custom needs.

HELIX defers operator pipelining and scheduling for asynchronous execution to Spark. Operators that can run concurrently are invoked in an arbitrary order, executed by Spark via Fair Scheduling. While by default we use Spark in the batch processing mode, it can be configured to perform stream processing using the same APIs as batch. We discuss optimizations for streaming in Chapter 5.

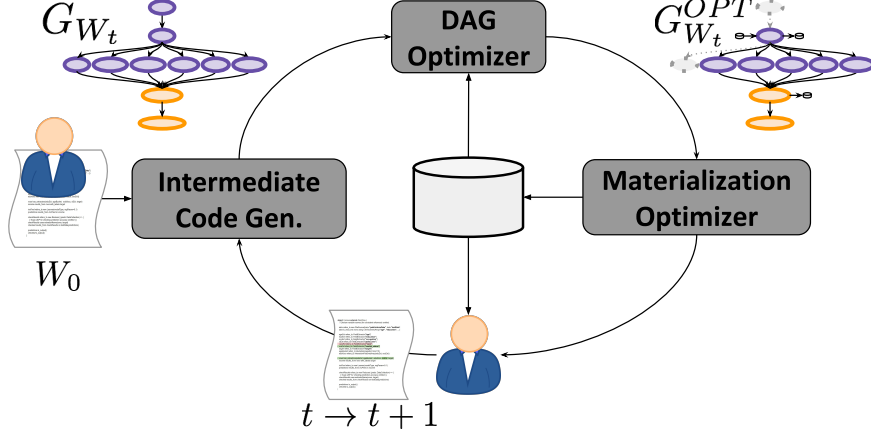


Figure 2.3: Roles of system components in the HELIX workflow lifecycle.

2.4 THE WORKFLOW LIFECYCLE

Given the system components described in the previous section, Figure 2.3 illustrates how they fit into the lifecycle of ML workflows. Starting with W_0 , an initial version of the workflow, the lifecycle includes the following stages:

1. **DAG Compilation.** The Workflow W_t is compiled into a DAG G_{W_t} of operator outputs.
2. **DAG Optimization.** The DAG optimizer creates a physical plan $G_{W_t}^{OPT}$ to be executed by pruning and ordering the nodes in G_{W_t} and deciding whether any computation can be replaced with loading previous results from disk.
3. **Materialization Optimization.** During execution, the materialization optimizer determines which nodes in $G_{W_t}^{OPT}$ should be persisted to disk for future use.
4. **User Interaction.** Upon execution completion, the user may modify the workflow from W_t to W_{t+1} based on the results. The updated workflow W_{t+1} fed back to HELIX marks the beginning of a new iteration, and the cycle repeats.

Without loss of generality, we assume that a workflow W_t is only executed once in each iteration. We model a repeated execution of W_t as a new iteration where $W_{t+1} = W_t$. Distinguishing two executions of the same workflow is important because they may have different run times—the second execution can reuse results materialized in the first execution for a potential run time reduction.

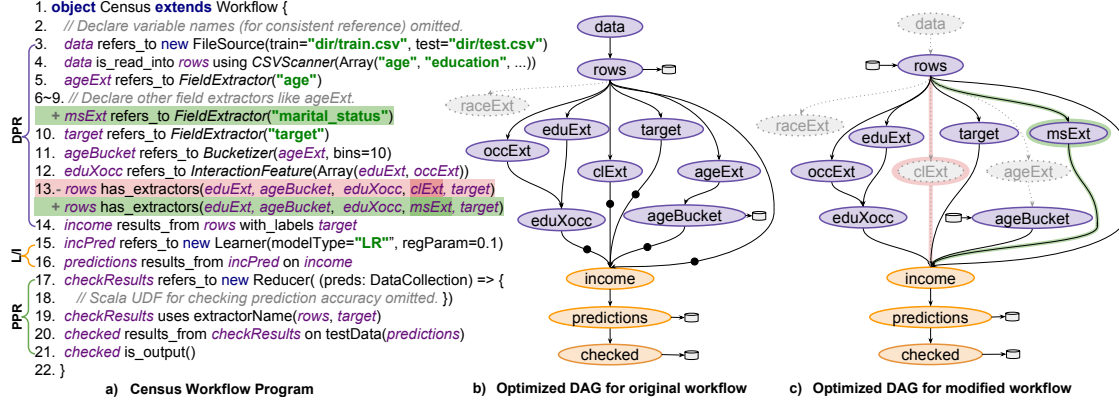


Figure 2.4: Example workflow for predicting income from census data.

2.5 EXAMPLE WORKFLOW

We demonstrate the usage of HELIX with a simple example ML workflow for predicting income using census data from Kohavi [22], shown in Figure 2.4a); this workflow will serve as a running example throughout the paper. Details about the individual operators will be provided in subsequent sections. We overlay the original workflow with an iterative update, with additions annotated with + and deletions annotated with −, while the rest of the lines are retained as is. We begin by describing the original workflow consisting of all the unannotated lines plus the line annotated with − (deletions).

Original Workflow: DPR Steps. First, after some variable name declarations, the user defines in line 3-4 a data collection rows read from a data source data consisting of two CSV files, one for training and one for test data, and names the columns of the CSV files age, education, etc. In lines 5-10, the user declares simple features that are values from specific named columns. Note that the user is not required to specify the feature type, which is automatically inferred by HELIX from data. In line 11 ageBucket is declared as a derived feature formed by discretizing age into ten buckets (whose boundaries are computed by HELIX), while line 12 declares an interaction feature, commonly used to capture higher-order patterns, formed out of the concatenation of eduExt and occExt.

Once the features are declared, the next step, line 13, declares the features to be extracted from and associated with each element of rows. Users do not need to worry about how these features are attached and propagated; users are also free to perform manual feature selection here, studying the impact of various feature combinations, by excluding some of the feature extractors. Finally, as last step of data preprocessing, line 14 declares that an example collection named income is to be made from rows using target as labels. Importantly, this step converts the features from human-readable formats (e.g., color=red) into an indexed

vector representation required for learning.

Original Workflow: L/I & PPR Steps. Line 15 declares an ML model named `incPred` with type “Logistic Regression” and regularization parameter 0.1, while line 16 specifies that `incPred` is to be learned on the training data in `income` and applied on all data in `income` to produce a new example collection called `predictions`. Line 17-18 declare a *Reducer* named `checkResults`, which outputs a scalar using a UDF for computing prediction accuracy. Line 19 explicitly specifies `checkResults`’s dependency on `target` since the content of the UDF is opaque to the optimizer. Line 20 declares that the output scalar named `checked` is only to be computed from the test data in `income`. Line 21 declares that `checked` must be part of the final output.

Original Workflow: Optimized DAG. The HELIX compiler first translates verbatim the program in Figure 2.4a) into a DAG, which contains all nodes including `raceExt` and all edges (including the dashed edge) except the ones marked with dots in Figure 2.4b). This DAG is then transformed by the optimizer, which prunes away `raceExt` (grayed out) because it does not contribute to the output, and adds the edges marked by dots to link relevant features to the model. DPR involves nodes in purple, and L/I and PPR involve nodes in orange. Nodes with a drum to the right are materialized to disk, either as mandatory output or for aiding in future iterations.

Updated Workflow: Optimized DAG. In the updated version of the workflow, a new feature named `msExt` is added (below line 9), and `clExt` is removed (line 13); correspondingly, in the updated DAG, a new node is added for `msExt` (green edges), while `clExt` gets pruned (pink edges). In addition, HELIX chooses to load materialized results for rows from the previous iteration allowing data to be pruned, avoiding a costly parsing step. HELIX also loads `ageBucket` instead of recomputing the bucket boundaries requiring a full scan. HELIX materializes predictions in both iterations since it has changed. Although predictions is not reused in the updated workflow, its materialization has high expected payoff over iterations because PPR iterations (changes to `checked` in this case) are the most common as per our survey results shown in Figure 2.1(c). This example illustrates that

1. Nodes selected for materialization lead to significant speedup in subsequent iterations.
2. HELIX reuses results safely, deprecating old results when changes are detected (e.g., predictions is not reused because of the model change).
3. HELIX correctly prunes away extraneous operations via dataflow analysis.

CHAPTER 3: PROGRAMMING INTERFACE

To program ML workflows with high-level abstractions, HELIX users program in a language called HML, an *embedded DSL* in Scala. An embedded DSL exists as a library in the host language (Scala in our case), leading to seamless integration. LINQ [23], a data query framework integrated in .NET languages, is another example of an embedded DSL. In HELIX, users can freely incorporate Scala code for user-defined functions (UDFs) directly into HML. JVM-based libraries can be imported directly into HML to support application-specific needs. Development in other languages can be supported with wrappers in the same style as PySpark [24].

3.1 OPERATIONS IN ML WORKFLOWS

In this section, we argue that common operations in ML workflows can be decomposed into a small set of *basis functions* \mathcal{F} . We first introduce \mathcal{F} and then enumerate its mapping onto operations in Scikit-learn [25], one of the most comprehensive ML libraries, thereby demonstrating coverage. In Section 3.2, we introduce HML, which implements the capabilities offered by \mathcal{F} .

As mentioned in Chapter 2, an ML workflow consists of three components: data preprocessing (DPR), learning/inference (L/I), and postprocessing (PPR). They are captured by the *Transformer*, *Estimator*, and *Predictor* interfaces in Scikit-learn, respectively. Similar interfaces can be found in many ML libraries, such as MLLib [26], TFX [27], and KeystoneML.

Data Representation. Conventionally, the input space to ML, \mathcal{X} , is a d -dimensional vector space, $\mathbb{R}^d, d \geq 1$, where each dimension corresponds to a feature. Each datapoint is represented by a feature vector (FV), $\mathbf{x} \in \mathbb{R}^d$. For notational convenience, we denote a d -dimensional FV, $\mathbf{x} \in \mathbb{R}^d$, as \mathbf{x}^d . While inputs in some applications can be easily loaded into FVs, e.g., images are 2D matrices that can be flattened into a vector, many others require more complex transformations, e.g., vectorization of text requires tokenization and word indexing. We denote the input dataset of FVs to an ML algorithm as \mathcal{D} .

DPR. The goal of DPR is to transform raw input data into \mathcal{D} . We use the term *record*, denoted by r , to refer to a data object in formats incompatible with ML, such as text and JSON, requiring preprocessing. Let $\mathcal{S} = \{r\}$ be a data source, e.g., a csv file, or a collection of text documents. DPR includes transforming records from one or more data sources from one format to another or into FVs $\mathbb{R}^{d'}$; as well as feature transformations (from \mathbb{R}^d to $\mathbb{R}^{d'}$).

DPR operations can thus be decomposed into the following categories:

- *Parsing* $r \mapsto (r_1, r_2, \dots)$: transforming a record into a set of records, e.g., parsing an article into words via *tokenization*.
- *Join* $(r_1, r_2, \dots) \mapsto r$: combining multiple records into a single record, where r_i can come from different data sources.
- *Feature Extraction* $r \mapsto \mathbf{x}^d$: extracting features from a record.
- *Feature Transformation* $T : \mathbf{x}^d \mapsto \mathbf{x}^{d'}$: deriving a new set of features from the input features.
- *Feature Concatenation* $(\mathbf{x}^{d_1}, \mathbf{x}^{d_2}, \dots) \mapsto \mathbf{x}^{\sum_i d_i}$: concatenating features extracted in separate operations to form an FV.

Note that sometimes these functions need to be *learned* from the input data. For example, discretizing a continuous feature x_i into four even-sized bins requires the distribution of x_i , which is usually estimated empirically by collecting all values of x_i in \mathcal{D} . We address this use case along with L/I next.

L/I. At a high-level, L/I is about learning a function f from the input \mathcal{D} , where $f : \mathcal{X} \rightarrow \mathbb{R}^{d'}$, $d' \geq 1$. This is more general than learning ML models, and also includes feature transformation functions mentioned above. The two main operations in L/I are 1) *learning*, which produces functions using data from \mathcal{D} , and 2) *inference*, which uses the function obtained from learning to draw conclusions about new data. Complex ML tasks can be broken down into simple learning steps captured by these two operations, e.g., image captioning can be broken down into object identification via classification, followed by sentence generation using a language model [15]. Thus, L/I can be decomposed into:

- *Learning* $\mathcal{D} \mapsto f$: learning a function f from the dataset \mathcal{D} .
- *Inference* $(\mathcal{D}, f) \mapsto \mathcal{Y}$: using the ML model f to infer feature values, i.e., *labels*, \mathcal{Y} from the input FVs in \mathcal{D} .

Note that labels can be represented as FVs like other features, hence the usage of a single \mathcal{D} in learning to represent both the training data and labels to unify the abstraction for both supervised and unsupervised learning and to enable easy model composition.

PPR. Finally, a wide variety of operations can take place in PPR, using the learned models and inference results from L/I as input, including model evaluation, data visualization, and other application-specific activities. The most commonly supported PPR operations

Scikit-learn DPR, L/I	Composed Members of \mathcal{F}
<code>fit(X[, y])</code>	<i>learning</i> ($\mathcal{D} \mapsto f$)
<code>predict_proba(X)</code>	<i>inference</i> ($((\mathcal{D}, f) \mapsto \mathcal{Y})$)
<code>predict(X)</code>	<i>inference</i> , optionally followed by <i>transformation</i>
<code>fit_predict(X[, y])</code>	<i>learning</i> , then <i>inference</i>
<code>transform(X)</code>	<i>transformation</i> or <i>inference</i> , depending on whether operation is learned via prior call to <code>fit</code>
<code>fit_transform(X)</code>	<i>learning</i> , then <i>inference</i>
Scikit-learn PPR	Composed Members of \mathcal{F}
<code>eval: score(y_{true}, y_{pred})</code>	<i>join</i> y_{true} and y_{pred} into a single dataset \mathcal{D} , then <i>reduce</i>
<code>eval: score(op, X, y)</code>	<i>inference</i> , then <i>join</i> , then <i>reduce</i>
<code>selection: fit(p₁, ..., p_n)</code>	<i>reduce</i> , implemented in terms of <i>learning</i> , <i>inference</i> , and <i>reduce</i> (for scoring)

Table 3.1: Scikit-learn DPR, L/I, and PPR coverage in terms of \mathcal{F} .

in general purpose ML libraries are model evaluation and model selection, which can be represented by a computation whose output does not depend on the size of the data \mathcal{D} . We refer to a computation with output sizes independent of input sizes as a *reduce*:

- *Reduce* (\mathcal{D}, s') $\mapsto s$: applying an operation on the input dataset \mathcal{D} and s' , where s' can be any non-dataset object. For example, s' can store a set of hyperparameters over which *reduce* optimizes, learning various models and outputting s , which can represent a function corresponding to the model with the best cross-validated hyperparameters.

3.1.1 Comparison with Scikit-learn

A dataset in Scikit-learn is represented as a matrix of FVs, denoted by X . This is conceptually equivalent to $\mathcal{D} = \{\mathbf{x}^d\}$ introduced earlier, as the order of rows in X is not relevant. Operations in Scikit-learn are categorized into dataset loading and transformations, learning, and model selection and evaluation [28]. Operations like loading and transformations that do not tailor their behavior to particular characteristics present in the dataset \mathcal{D} map trivially onto the DPR basis functions $\in \mathcal{F}$ introduced at the start of Section 3.1, so we focus on comparing data-dependent DPR and L/I, and model selection and evaluation.

Scikit-learn Operations for DPR and L/I. Scikit-learn objects for DPR and L/I implement one or more of the following interfaces [29]:

- **Estimator**, used to indicate that an operation has data-dependent behavior via a `fit(X[, y])` method, where `X` contains FVs or raw records, and `y` contains labels if the operation represents a supervised model.
- **Predictor**, used to indicate that the operation may be used for inference via a `predict(X)` method, taking a matrix of FVs and producing predicted labels. Additionally, if the operation implementing Predictor is a classifier for which inference may produce raw floats (interpreted as probabilities), it may optionally implement `predict_proba`.
- **Transformer**, used to indicate that the operation may be used for feature transformations via a `transform(X)` method, taking a matrix of FVs and producing a new matrix `Xnew`.

An operation implementing both Estimator and Predictor has a `fit_predict` method, and an operation implementing both Estimator and Transformer has a `fit_transform` method, for when inference or feature transformation, respectively, is applied immediately after fitting to the data. The rationale for providing a separate Estimator interface is likely due to the fact that it is useful for both feature transformation and inference to have data-dependent behavior determined via the result of a call to `fit`. For example, a useful data-dependent feature transformation for a Naive Bayes classifier maps word tokens to positions in a sparse vector and tracks word counts. The position mapping will depend on the vocabulary represented in the raw training data. Other examples of data-dependent transformations include feature scaling, discretization, imputation, dimensionality reduction, and kernel transformations.

Coverage in terms of basis functions \mathcal{F} . The first part of Table 3.1 summarizes the mapping from Scikit-learn’s interfaces for DPR and L/I to (compositions of) basis functions from \mathcal{F} . In particular, note that there is nothing special about Scikit-learn’s use of separate interfaces for inference (via Predictor) and data-dependent transformations (via Transformer); the separation exists mainly to draw attention to the semantic separation between DPR and L/I.

Scikit-learn Operations for PPR. Scikit-learn interfaces for operations implementing model selection and evaluation are not as standardized as those for DPR and L/I. For evaluation, the typical strategy is to define a simple function that compares model outputs with labels, computing metrics like accuracy or F_1 score. For model selection, the typical strategy is to define a class that implements methods `fit` and `score`. The `fit` method takes a set of hyperparameters over which to search, with different models scored according to

the score method (with identical interface as for evaluation in Scikit-learn). The actual model over which hyperparameter search is performed is implemented by an Estimator that is passed into the model selection operation’s constructor.

Coverage in terms of basis functions \mathcal{F} . As summarized in the second part of Table 3.1, Scikit-learn’s operations for evaluation may be implemented via compositions of (optionally) *inference*, *joining*, and *reduce* $\in \mathcal{F}$. Model selection may be implemented via a *reduce* that internally uses learning basis functions to learn models for the set of hyperparameters specified by s' , followed by composition with *inference* and another *reduce* $\in \mathcal{F}$ for scoring, eventually returning the final selected model.

3.2 HML

HML is a declarative language for specifying an ML workflow DAG. The basic building blocks of HML are HELIX *objects*, which correspond to the nodes in the DAG. Each HELIX object is either a *data collection* (DC) or an *operator*. Statements in HML either declare new instances of objects or relationships between declared objects. Users program the entire workflow in a single **Workflow** interface, as shown in Figure 2.4a). The complete grammar for HML in Backus-Naur Form is shown in Figure 3.1, and the semantics of all of the expressions is shown in Table 3.2. Here, we describe high-level concepts including DCs and operators and discuss the strengths and limitations of HML in Section 3.3.

3.2.1 Data Collections

A *data collection* (DC) is analogous to a relation in a RDBMS; each *element* in a DC is analogous to a tuple. The content of a DC either derives from disk, e.g., data in Line 3 in Figure 2.4a), or from operations on other DCs, e.g., rows in Line 4 in Figure 2.4a). An element in a DC can either be a *semantic unit*, the data structure for DPR, or an *example*, the data structure for L/I.

A DC can only contain a single type of element. DC_{SU} and DC_E denote a DC of semantic units and a DC of examples, respectively. The type of elements in a DC is determined by the operator that produced the DC and not explicitly specified by the user. We elaborate on the relationship between operators and element types in Section 3.2.2, after introducing the operators.

Semantic units. Recall that many DPR operations require going through the entire dataset to learn the exact transformation or extraction function. For a workflow with many such

```

⟨var⟩      ::= ⟨string⟩
⟨scanner⟩  ::= ⟨var⟩ | ⟨scanner-obj⟩
⟨extractor⟩ ::= ⟨var⟩ | ⟨extractor-obj⟩
⟨typed-ext⟩ ::= '(' ⟨var⟩ ',' ⟨extractor⟩ ')'
⟨extractors⟩ ::= '(' ⟨extractor⟩ { ',' ⟨extractor⟩ } ')'
⟨typed-exts⟩ ::= '(' ⟨typed-ext⟩ { ',' ⟨typed-ext⟩ } ')'
⟨obj⟩      ::= ⟨data-source⟩ | ⟨scanner-obj⟩ | ⟨extractor-obj⟩ |
               ⟨learner-obj⟩ | ⟨synthesizer-obj⟩ | ⟨reducer-obj⟩

⟨assign⟩   ::= ⟨var⟩ 'refers_to' ⟨obj⟩
⟨expr1⟩    ::= ⟨var⟩ 'is_read_into' ⟨var⟩ 'using' ⟨scanner⟩
⟨expr2⟩    ::= ⟨var⟩ 'has_extractors' ⟨extractors⟩
⟨list⟩     ::= ⟨var⟩ | '(' ⟨var⟩ ',' ⟨var⟩ { ',' ⟨var⟩ } ')'
⟨apply⟩    ::= ⟨var⟩ 'on' ⟨list⟩
⟨expr3⟩    ::= ⟨apply⟩ 'as_examples' ⟨var⟩
⟨expr4⟩    ::= ⟨apply⟩ 'as_results' ⟨var⟩
⟨expr5⟩    ::= ⟨var⟩ 'as_examples' ⟨var⟩
               'with_labels' ⟨extractor⟩
⟨expr6⟩    ::= ⟨var⟩ 'uses' ⟨typed-exts⟩
⟨expr7⟩    ::= ⟨var⟩ 'is_output()'
⟨statement⟩ ::= ⟨assign⟩ | ⟨expr1⟩ | ⟨expr2⟩ | ⟨expr3⟩ | ⟨expr4⟩ |
               ⟨expr5⟩ | ⟨expr6⟩ | ⟨expr7⟩ | ⟨Scala expr⟩
⟨program⟩  ::= 'object' ⟨string⟩ 'extends Workflow {'
               { ⟨statement⟩ ⟨line-break⟩ }
               '}'

```

Figure 3.1: HELIX syntax in Extended Backus-Naur Form. $\langle string \rangle$ denotes a legal String object in Scala; $\langle *-obj \rangle$ denotes the correct syntax for instantiating object of type $*$; $\langle Scala\ expr \rangle$ denotes any legal Scala expression. A HELIX Workflow can be comprised of any combination of HELIX and Scala expressions, a direct benefit of being an embedded DSL.

Phrase	Usage	Operation	Example
refers_to	<i>string</i> refers_to HELIX <i>object</i>	Register a HELIX <i>object</i> to a <i>string</i> name	“ext1” refers_to Extractor(...)
is_read_into ... using	$DC_i[SU]$ is read into $DC_j[SU]$ using <i>scanner</i>	Apply <i>scanner</i> on DC_i to obtain DC_j	“sentence” is_read_into “word” using whitespaceTokenizer
has_extractors	$DC[SU]$ has_extractors <i>extractor</i> +	Apply extractors to DC	“word” has_extractors (“ext1”, “ext2”)
on	<i>synthesizer/learner/reducer</i> on $DC[*]$ +	Apply <i>synthesizer/learner</i> on input $DC(s)$ to produce an output $DC[E]$	“match” on (“person_candidate”, “known_persons”)
results_from	$DC_i[E]$ results_from $DC_j[*]$ [with_label <i>extractor</i>]	Wrap each element in DC_i in an Example and optionally labels the Examples with the output of <i>extractor</i> .	“income” results_from “rows” with_label “target”
uses	$DC[E]$ /Scalar results_from <i>clause</i>	Specify the name for <i>clause</i> ’s output $DC[E]$.	“learned” results_from “L” on “income”
	<i>synthesizer/learner/reducer</i> uses <i>extractors</i> +	Specify <i>synthesizer/learner</i> ’s dependency on the output of <i>extractors</i> + to prevent pruning or uncaching of intermediate results due to optimization.	“match” uses (“ext1”, “ext2”)
is_output	$DC[*]$ /result is_output	Requires $DC/result$ to be materialized.	“learned” is_output

Table 3.2: Usage and functions of key phrases in HML. $DC[A]$ denotes a DC with name DC and elements of type $A \in \{SU, E\}$, with $A = *$ indicating both types are legal. $x+$ indicates that x appears one or more times. When appearing in the same statement, on takes precedence over $results_from$.

operations, processing \mathcal{D} to learn each operator separately can be highly inefficient. We introduce the notion of semantic units (SU) to compartmentalize the logical and physical representations of features, so that the learning of DPR functions can be delayed and batched.

Formally, each SU contains an input i , which can be a set of records or FVs, a pointer to a DPR function f , which can be of type parsing, join, feature extraction, feature transformation, or feature concatenation, and an output o , which can be a set of records or FVs and is the output of f on i . The variables i and f together serve as the *semantic*, or logical, representation of the features, whereas o is the lazily evaluated physical representation that can only be obtained after f is fully instantiated.

Examples. Examples gather all the FVs contained in the output of various SUs into a single FV for learning. Formally, an example contains a set of SUs S , and an optional pointer to one of the SUs whose output will be used as the label in supervised settings, and an output FV, which is formed by concatenating the outputs of S . In the implementation, the order of SUs in the concatenation is determined globally across \mathcal{D} , and SUs whose outputs are not FVs are filtered out.

Sparse vs. Dense Features. The combination of SUs and examples affords HELIX a great deal of flexibility in the physical representation of features. Users can explicitly program their DPR functions to output dense vectors, in applications such as computer vision. For sparse categorical features, they are kept in the raw key-value format until the final FV assembly, where they are transformed into sparse or dense vectors depending on whether the ML algorithm supports sparse representations. Note that users do not have to commit to a single representation for the entire application, since different SUs can contain different types of features. When assembling a mixture of dense and sparse FVs, HELIX currently opts for a dense representation but can be extended to support optimizations considering space and time tradeoffs.

Unified learning support. HML provides unified support for training and test data by treating them as a single DC, as done in Line 4 in Figure 2.4a). This design ensures that both training and test data undergo the exact same data preprocessing steps, eliminating bugs caused by inconsistent data preprocessing procedures handling training and test data separately. HELIX automatically selects the appropriate data for training and evaluation. However, if desired, users can handle training and test data differently by specifying separate DAGs for training and testing. Common operators can be shared across the two DAGs without code duplication.

3.2.2 Operators

Operators in HELIX are designed to cover the functions enumerated in Section 3.1, using the data structures introduced above. A HELIX *operator* takes one or more DCs and outputs DCs, ML models, or scalars. Each operator encapsulates a function f , written in Scala, to be applied to individual elements in the input DCs. As noted above, f can be learned from the input data or user defined. Like in Scikit-learn, HML provides off-the-shelf implementations for common operations for ease of use. We describe the relationships between operator interfaces in HML and \mathcal{F} enumerated in Section 3.1 below.

Scanner. *Scanner* is the interface for parsing $\in \mathcal{F}$ and acts like a flatMap, i.e., for each input element, it adds zero or more elements to the output DC. Thus, it can also be used to perform filtering. The input and output of Scanner are DC_{SUs} . CSVScanner in Line 4 of Figure 2.4a) is an example of a Scanner that parses lines in a CSV file into key-value pairs for columns.

Synthesizer. *Synthesizer* supports join $\in \mathcal{F}$, for elements both across multiple DCs and within the same DC. Thus, it can also support aggregation operations such as sliding windows in time series. Synthesizers also serve the important purpose of specifying the set of SUs that make up an example (where output FVs from the SUs are automatically assembled into a single FV). In the simple case where each SU in a DC_{SU} corresponds to an example, a pass-through synthesizer is implicitly declared by naming the output DC_E , such as in Line 14 of Figure 2.4a).

Learner. *Learner* is the interface for learning and inference $\in \mathcal{F}$, in a single operator. A learner operator L contains a learned function f , which can be populated by learning from the input data or loading from disk. f can be an ML model, but it can also be a feature transformation function that needs to be learned from the input dataset. When f is empty, L learns a model using input data designated for model training; when f is populated, L performs inference on the input data using f and outputs the inference results into a DC_E . For example, the learner incPred in Line 15 of Figure 2.4a) is a learner trained on the “train” portion of the DC_E income and outputs inference results as the DC_E predictions.

Extractor. *Extractor* is the interface for feature extraction and feature transformation $\in \mathcal{F}$. Extractor contains the function f applied on the input of SUs, thus the input and output to an extractor are DC_{SUs} . For functions that need to be learned from data, Extractor contains a pointer to the learner operator for learning f .

Reducer. Reducer is the interface for reduce $\in \mathcal{F}$ and thus the main operator interface for PPR. The inputs to a reducer are DC_E and an optional scalar and the output is a scalar, where scalars refer to non-dataset objects. For example, checkResults in Figure 2.4a) Line

17 is a reducer that computes the prediction accuracy of the inference results in predictions.

3.3 SCOPE AND LIMITATIONS

Coverage. In Section 3.1, we described how the set of basis operations \mathcal{F} we propose covers all major operations in Scikit-learn, one of the most comprehensive ML libraries. We then showed in Section 3.2 that HML captures all functions in \mathcal{F} . While HML’s interfaces are general enough to support all the common use cases, users can additionally manually plug into our interfaces external implementations, such as from MLLib [26] and Weka [30], of missing operations. *Note that we provide utility functions that allow functions to work directly with raw records and FVs instead of HML data structures to enable direct application of external libraries.* For example, since all MLLib models implement the train (equivalent to learning) and predict (equivalent to inference) methods, they can easily be plugged into Learner in HELIX. We demonstrate in Chapter 6 that the current set of implemented operations is sufficient for supporting applications across different domains.

Limitations. Since HELIX currently relies on its Scala DSL for workflow specification, popular non-JVM libraries, such as TensorFlow [31] and Pytorch [32], cannot be imported easily without significantly degrading performance compared to their native runtime environment. Developers with workflows implemented in other languages will need to translate them into HML, which should be straightforward due to the natural correspondence between HELIX operators and those in standard ML libraries, as established in Section 3.2. That said, our contributions in materialization and reuse apply across all languages. In the future, we plan on abstracting the DAG representation in HELIX into a language-agnostic system that can sit below the language layer for all DAG based systems, including TensorFlow, Scikit-learn, and Spark.

The other downside of HML is that ML models are treated largely as black boxes. Thus, work on optimizing learning, e.g., [33, 34], orthogonal to (and can therefore be combined with) our work, which operates at a coarser granularity.

CHAPTER 4: COMPILATION AND REPRESENTATION

In this section, we describe the **Workflow DAG**, the abstract model used internally by HELIX to represent a **Workflow** program. The **Workflow DAG** model enables operator-level change tracking between iterations and end-to-end optimizations.

4.1 THE WORKFLOW DAG

At compile time, HELIX’s intermediate code generator constructs a *Workflow DAG* from HML declarations, with nodes corresponding to operator outputs, (DCs, scalars, or ML models), and edges corresponding to input-output relationships between operators.

Definition 4.1 *For a Workflow W containing HELIX operators $F = \{f_i\}$, the Workflow DAG is a directed acyclic graph $G_W = (N, E)$, where node $n_i \in N$ represents the output of $f_i \in F$ and $(n_i, n_j) \in E$ if the output of f_i is an input to f_j .*

Figure 2.4b) shows the **Workflow DAG** for the program in Figure 2.4a). Nodes for operators involved in DPR are colored purple whereas those involved in L/I and PPR are colored orange. This transformation is straightforward, creating a node for each declared operator and adding edges between nodes based on the linking expressions, e.g., `A results from B` creates an edge (B, A) . Additionally, the intermediate code generator introduces edges not specified in the **Workflow** between the extractor and the synthesizer nodes, such as the edges marked by dots (\bullet) in Figure 2.4b). These edges connect extractors to downstream DCs in order to automatically aggregate all features for learning. One concern is that this may lead to redundant computation of unused features; we describe pruning mechanisms to address this issue in Section 5.4.

4.2 TRACKING CHANGES

As described in Section 2.4, a user starts with an initial workflow W_0 and iterates on this workflow. Let W_t be the version of the workflow at iteration $t \geq 0$ with the corresponding DAG $G_W^t = (N_t, E_t)$; W_{t+1} denotes the workflow obtained in the next iteration. To describe the changes between W_t and W_{t+1} , we introduce the notion of *equivalence*.

Definition 4.2 *A node $n_i^t \in N_t$ is equivalent to $n_i^{t+1} \in N_{t+1}$, denoted as $n_i^t \equiv n_i^{t+1}$, if **a)** the operators corresponding to n_i^t and n_i^{t+1} compute identical results on the same inputs and*

b) $n_j^t \equiv n_j^{t+1} \forall n_j^t \in \text{parents}(n_i^t), n_j^{t+1} \in \text{parents}(n_i^{t+1})$. We say $n_i^{t+1} \in N_{t+1}$ is original if it has no equivalent node in N_t .

Equivalence is symmetric, i.e., $n_i^{t'} \equiv n_i^t \Leftrightarrow n_i^t \equiv n_i^{t'}$, and transitive, i.e., $(n_i^t \equiv n_i^{t'} \wedge n_i^{t'} \equiv n_i^{t''}) \Rightarrow n_i^t \equiv n_i^{t''}$. Newly added operators in W_{t+1} do not have equivalent nodes in W_t ; neither do nodes in W_t that are removed in W_{t+1} . For a node that persists across iterations, we need both the operator and the ancestor nodes to stay the same for equivalence. Using this definition of equivalence, we determine if intermediate results on disk can be safely reused through the notion of equivalent materialization:

Definition 4.3 A node $n_i^t \in N_t$ has an equivalent materialization if $n_i^{t'}$ is stored on disk, where $t' \leq t$ and $n_i^{t'} \equiv n_i^t$.

One challenge in determining equivalence is deciding whether two versions of an operator compute the same results on the same input. For arbitrary functions, this is undecidable as proven by Rice’s Theorem [35]. The programming language community has a large body of work on verifying operational equivalence for specific classes of programs [36, 37, 38]. HELIX currently employs a simple representational equivalence verification—an operator remains equivalent across iterations if its declaration in the DSL is not modified and all of its ancestors are unchanged. Incorporating more advanced techniques for verifying equivalence is future work.

To guarantee correctness, i.e., results obtained at iteration t reflect the specification for W_t and are computed from the appropriate input, we impose the constraint:

Constraint 4.1 At iteration $t + 1$, if an operator n_i^{t+1} is original, it must be recomputed.

With Constraint 4.1, our current approach to tracking changes yields the following guarantee on result correctness:

Theorem 4.1 HELIX returns the correct results if the changes between iterations are made only within the programming interface, i.e., all other factors, such as library versions and files on disk, stay invariant, i.e., unchanged, between executions at iteration t and $t + 1$.

First, note that the results for W_0 are correct since there is no reuse at iteration 0. Suppose for contradiction that given the results at t are correct, the results at iteration $t + 1$ are incorrect, i.e., $\exists n_i^{t+1}$ s.t. the results for n_i^t are reused when n_i^{t+1} is original. Under the invariant conditions in Theorem 4.1, we can only have $n_i^{t+1} \neq n_i^t$ if the code for n_i changed or the code changed for an ancestor of n_i . Since HELIX detects all code changes, it identifies all original operators. Thus, for the results to be incorrect in HELIX, we must have reused n_i^t for some original n_i^{t+1} . However, this violates Constraint 4.1. Therefore, the results for W_t are correct $\forall t \geq 0$.

CHAPTER 5: OPTIMIZATION

In this section, we describe HELIX’s workflow-level optimizations, motivated by the observation that *workflows often share a large amount of intermediate computation between iterations*; thus, if certain intermediate results are materialized at iteration t , these can be used at iteration $t + 1$. We identify two distinct sub-problems: OPT-EXEC-PLAN, which selects the operators to reuse given previous materializations (Section 5.2), and OPT-MAT-PLAN, which decides what to materialize to accelerate future iterations (Section 5.3). We finally discuss pruning optimizations to eliminate redundant computations (Section 5.4). We begin by introducing common notation and definitions.

5.1 PRELIMINARIES

When introducing variables below, we drop the iteration number t from W_t and G_W^t when we are considering a static workflow.

Operator Metrics. In a Workflow DAG $G_W = (N, E)$, each node $n_i \in N$ corresponding to the output of the operator f_i is associated with a compute time c_i , the time it takes to compute n_i from inputs in memory. Once computed, n_i can be materialized on disk and loaded back in subsequent iterations in time l_i , referred to as its *load time*. If n_i does not have an equivalent materialization as defined in Definition 4.3, we set $l_i = \infty$. Root nodes in the Workflow DAG, which correspond to data sources, have $l_i = c_i$.

Operator State. During the execution of workflow W , each node n_i assumes one of the following states:

- *Load*, or S_l , if n_i is loaded from disk;
- *Compute*, or S_c , n_i is computed from inputs;
- *Prune*, or S_p , if n_i is skipped (neither loaded nor computed).

Let $s(n_i) \in \{S_l, S_c, S_p\}$ denote the state of each $n_i \in N$. To ensure that nodes in the Compute state have their inputs available, i.e., not *pruned*, the states in a Workflow DAG $G_W = (N, E)$ must satisfy the following *execution state constraint*:

Constraint 5.1 *For a node $n_i \in N$, if $s(n_i) = S_c$, then $s(n_j) \neq S_p$ for every $n_j \in \text{parents}(n_i)$.*

Workflow Run Time. A node n_i in state S_c , S_l , or S_p has run time c_i , l_i , or 0, respectively. The total run time of W w.r.t. s is thus

$$T(W, s) = \sum_{n_i \in N} \mathbb{I}\{s(n_i) = S_c\} c_i + \mathbb{I}\{s(n_i) = S_l\} l_i \quad (5.1)$$

where $\mathbb{I}\{\cdot\}$ is the indicator function.

Clearly, setting all nodes to S_p trivially minimizes Equation 5.1. However, recall that Constraint 4.1 requires all original operators to be rerun. Thus, if an original operator n_i is introduced, we must have $s(n_i) = S_c$, which by Constraint 5.1 requires that $S(n_j) \neq S_p \forall n_j \in \text{parents}(n_i)$. Deciding whether to load or compute the parents can have a cascading effect on the states of their ancestors. We explore how to determine the states for each nodes to minimize Equation 5.1 next.

5.2 OPTIMAL EXECUTION PLAN

The *Optimal Execution Plan* (OEP) problem is the core problem solved by HELIX’s DAG optimizer, which determines at compile time the optimal execution plan given results and statistics from previous iterations.

Problem 5.1 (OPT-EXEC-PLAN) *Given a Workflow W with DAG $G_W = (N, E)$, the compute time and the load time c_i, l_i for each $n_i \in N$, and a set of previously materialized operators M , find a state assignment $s : N \rightarrow \{S_c, S_l, S_p\}$ that minimizes $T(W, s)$ while satisfying Constraint 4.1 and Constraint 5.1.*

Let $T^*(W)$ be the minimum execution time achieved by the solution to OEP, i.e.,

$$T^*(W) = \min_s T(W, s) \quad (5.2)$$

Since this optimization takes place *prior* to execution, we must resort to operator statistics from past iterations. *This does not compromise accuracy because if a node n_i has an equivalent materialization as defined in Definition 4.2, we would have run the exact same operator before and recorded accurate c_i and l_i .* A node n_i without an equivalent materialization, such as a model with changed hyperparameters, needs to be recomputed (Constraint 4.1).

Deciding to load certain nodes can have cascading effects since ancestors of a loaded node can potentially be pruned, leading to large reductions in run time. On the other hand, Constraint 5.1 disallows the parents of computed nodes to be pruned. Thus, the decisions to load a node n_i can be affected by nodes outside of the set of ancestors to n_i . For example, in

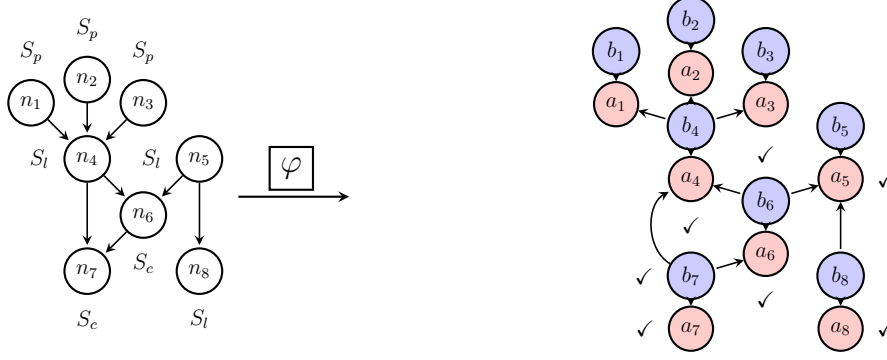


Figure 5.1: Transforming a Workflow DAG to a set of projects and dependencies. Checkmarks (✓) in the RHS DAG indicate a feasible solution to PSP, which maps onto the node states (S_p, S_c, S_l) in the LHS DAG.

the DAG on the left in Figure 5.1, loading n_7 allows n_{1-6} to be potentially pruned. However, the decision to compute n_8 , possibly arising from the fact that $l_8 \gg c_8$, requires that n_5 must not be pruned.

Despite such complex dependencies between the decisions for individual nodes, Problem 5.1 can be solved optimally in polynomial time through a linear time reduction to the *project-selection problem* (PSP), which is an application of MAX-FLOW [8].

Problem 5.2 PROJ-SELECTION-PROBLEM (PSP) *Let P be a set of projects. Each project $i \in P$ has a real-valued profit p_i and a set of prerequisites $Q \subseteq P$. Select a subset $A \subseteq P$ such that all prerequisites of a project $i \in A$ are also in A and the total profit of the selected projects, $\sum_{i \in A} p_i$, is maximized.*

Reduction to the Project Selection Problem. We can reduce an instance of Problem 5.1 x to an equivalent instance of PSP y such that the optimal solution to y maps to an optimal solution of x . Let $G = (N, E)$ be the Workflow DAG in x , and P be the set of projects in y . We can visualize the prerequisite requirements in y as a DAG with the projects as the nodes and an edge (j, i) indicating that project i is a prerequisite of project j . The reduction, φ , depicted in Figure 5.1 for an example instance of x , is shown in Algorithm 5.1. For each node $n_i \in N$, we create two projects in PSP: a_i with profit $-l_i$ and b_i with profit $l_i - c_i$. We set a_i as the prerequisite for b_i . For an edge $(n_i, n_j) \in E$, we set the project a_i corresponding to node n_i as the prerequisite for the project b_j corresponding to node n_j . Selecting both projects a_i and b_i corresponding to n_i is equivalent to computing n_i , i.e., $s(n_i) = S_c$, while selecting only a_i is equivalent to loading n_i , i.e., $s(n_i) = S_l$. Nodes with neither projects selected are pruned. An example solution mapping from PSP to OEP is shown in Figure 5.1. Projects $a_4, a_5, a_6, b_6, a_7, b_7, a_8$ are selected, which cause nodes n_4, n_5, n_8 to be loaded, n_6 and n_7 to be computed, and n_1, n_2, n_3 to be pruned.

Algorithm 5.1: OEP via Reduction to PSP

Input: $G_W = (N, E), \{l_i\}, \{c_i\}$

```
1  $P \leftarrow \emptyset;$ 
2 for  $n_i \in N$  do
3    $P \leftarrow P \cup \{a_i\};$  // Create a project  $a_i$ 
4    $profit[a_i] \leftarrow -l_i;$  // Set profit of  $a_i$  to  $-l_i$ 
5    $P \leftarrow P \cup \{b_i\};$  // Create a project  $b_i$ 
6    $profit[b_i] \leftarrow l_i - c_i;$  // Set profit of  $b_i$  to  $l_i - c_i$ 
   // Add  $a_i$  as prerequisite for  $b_i$ .;
7    $prerequisite[b_i] \leftarrow prerequisite[b_i] \cup a_i;$ 
8   for  $(n_i, n_j) \in \{\text{edges leaving from } n_i\} \subseteq E$  do
   // Add  $a_i$  as prerequisite for  $b_j$ .;
9    $prerequisite[b_j] \leftarrow prerequisite[b_j] \cup a_i;$ 
   //  $A$  is the set of projects selected by PSP;
10  $A \leftarrow \text{PSP}(P, profit[], prerequisite[]);$ 
11 for  $n_i \in N$  do // Map PSP solution to node states
12   if  $a_i \in A$  and  $b_i \in A$  then
13      $s[n_i] \leftarrow S_c;$ 
14   else if  $a_i \in A$  and  $b_i \notin A$  then
15      $s[n_i] \leftarrow S_l;$ 
16   else
17      $s[n_i] \leftarrow S_p;$ 
18 return  $s[];$  // State assignments for nodes in  $G_W$ .
```

Overall, the optimization objective in PSP models the “savings” in OEP incurred by loading nodes instead of computing them from inputs. We create an equivalence between cost minimization in OEP and profit maximization in PSP by mapping the costs in OEP to negative profits in PSP. For a node n_i , picking only project a_i (equivalent to loading n_i) has a profit of $-l_i$, whereas picking both a_i and b_i (equivalent to computing n_i) has a profit of $-l_i + (l_i - c_i) = -c_i$. The prerequisites established in Line 7 that require a_i to also be picked if b_i is picked are to ensure correct cost to profit mapping. The prerequisites established in Line 9 corresponds to Constraint 5.1. For a project b_i to be picked, we must pick every a_j corresponding to each parent n_j of n_i . If it is impossible ($l_j = \infty$) or costly to load n_j , we can offset the load cost by picking b_j for computing n_j . However, computing n_j also requires its parents to be loaded or computed, as modeled by the outgoing edges from b_j . The fact that a_i projects have no outgoing edges corresponds to the fact loading a node removes its dependency on all ancestor nodes.

Theorem 5.1 *Given an instance of OPT-EXEC-PLAN x , the reduction in Algorithm 5.1 produces a feasible and optimal solution to x .*

For clarity, we first formulate OPT-EXEC-PLAN as an integer linear program before presenting the proof itself.

Integer Linear Programming Formulation. Problem 5.1 can be formulated as an integer linear program (ILP) as follows. First, for each node $n_i \in G$, introduce binary indicator variables X_{a_i} and X_{b_i} defined as follows:

$$\begin{aligned} X_{a_i} &= \mathbb{I}\{s(n_i) \neq S_p\} \\ X_{b_i} &= \mathbb{I}\{s(n_i) = S_c\} \end{aligned}$$

That is, $X_{a_i} = 1$ if node n_i is not pruned, and $X_{b_i} = 1$ if node n_i is computed. Note that it is not possible to have $X_{a_i} = 0$ and $X_{b_i} = 1$. Also note that these variables uniquely determine node n_i 's state $s(n_i)$.

With the $\{X_{a_i}\}$ and $\{X_{b_i}\}$ thus defined, our ILP is as follows:

$$\begin{aligned} &\underset{X_{a_i}, X_{b_i}}{\text{minimize}} && \sum_{i=1}^{|N|} X_{a_i} l_i + X_{b_i} (c_i - l_i) \end{aligned} \tag{5.3a}$$

$$\text{subject to} \quad X_{a_i} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \tag{5.3b}$$

$$\sum_{n_j \in \text{Pa}(n_i)} X_{a_j} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \tag{5.3c}$$

$$X_{a_i}, X_{b_i} \in \{0, 1\}, \quad 1 \leq i \leq |N| \tag{5.3d}$$

Constraint 5.3b prevents the assignment $X_{a_i} = 0$ (n_i is pruned) and $X_{b_i} = 1$ (n_i is computed), since a pruned node cannot also be computed by definition. Constraint 5.3c is equivalent to Constraint 5.1 — if $X_{b_i} = 1$ (n_i is computed), any parent n_j of n_i must not be pruned, i.e., $X_{a_j} = 1$, in order for the sum to be nonnegative. Constraint 5.3d requires the solution to be integers.

This formulation does not specify a constraint corresponding to Constraint 4.1. Instead, we enforce Constraint 4.1 by setting the load and compute costs of nodes that need to be recomputed to specific values, as inputs to Problem 5.1. Specifically, we set the load cost to ∞ and the compute cost to $-\epsilon$ for a small $\epsilon > 0$. With these values, the cost of a node in S_l, S_p, S_c are $\infty, 0, -\epsilon$ respectively, which makes S_c a clear choice for minimizing Eq(5.3a).

Although ILPs are, in general, NP-Hard, the astute reader may notice that the constraint matrix associated with the above optimization problem is *totally unimodular* (TU), which means that an optimal solution for the LP-relaxation (which removes constraint 5.3d in the problem above) assigns integral values to $\{X_{a_i}\}$ and $\{X_{b_i}\}$, indicating that it is both optimal and feasible for the problem above as well [39]. In fact, it turns out that the above problem is

the dual of a flow problem; specifically, it is a minimum cut problem [40, 41]. This motivates the reduction introduced in Section 5.2.

Main proof. The proof for Theorem 5.1 follows directly from the two lemmas proven below. Recall that given an optimal solution A to PSP, we obtain the optimal state assignments for OEP using the following mapping:

$$s(n_i) = \begin{cases} S_c & \text{if } a_i \in A \text{ and } b_i \in A \\ S_l & \text{if } a_i \in A \text{ and } b_i \notin A \\ S_p & \text{if } a_i \notin A \text{ and } b_i \notin A \end{cases} \quad (5.4)$$

Lemma 5.1 *A feasible solution to PSP under φ also produces a feasible solution to OEP.*

We first show that satisfying the prerequisite constraint in PSP leads to satisfying Constraint 5.1 in OPT-EXEC-PLAN. Suppose for contradiction that a feasible solution to PSP under φ does not produce a feasible solution to OEP. This implies that for some node $n_i \in N$ s. t. $s(n_i) = S_c$, at least one parent n_j has $s(n_j) = S_p$. By the inverse of Eq (5.4), $s(n_i) = S_c$ implies that b_i was selected, while $s(n_j) = S_p$ implies that neither a_j nor b_j was selected. By construction, there exists an edge $a_j \rightarrow b_i$. The project selection entailed by the operator states leads to a violation of the prerequisite constraint. Thus, a feasible solution to PSP must produce a feasible solution to OEP under φ .

Lemma 5.2 *An optimal solution to PSP is also an optimal solution to OEP under φ .*

Let Y_{a_i} be the indicator for whether project a_i is selected, Y_{b_i} for the indicator for b_i , and $p(x_i)$ be the profit for project x_i . The optimization object for PSP can then be written as

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} Y_{a_i} p(a_i) + Y_{b_i} p(b_i) \quad (5.5)$$

Substituting our choice for $p(a_i)$ and $P(b_i)$, Eq (5.5) becomes

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} -Y_{a_i} l_i + Y_{b_i} (l_i - c_i) \quad (5.6)$$

$$= \max_{Y_{a_i}, Y_{b_i}} - \sum_{i=1}^{|N|} (Y_{a_i} - Y_{b_i}) l_i + Y_{b_i} c_i \quad (5.7)$$

The mapping established by Eq (5.4) is equivalent to setting $X_{a_i} = Y_{a_i}$ and $X_{b_i} = Y_{b_i}$.

Thus the maximization problem in Eq (5.7) is equivalent to the minimization problem in Eq (5.3a), and we obtain an optimal solution to OEP from the optimal solution to PSP.

Computational Complexity. For a Workflow DAG $G_W = (N_W, E_W)$ in OEP, the reduction above results in $\mathcal{O}(|N_W|)$ projects and $\mathcal{O}(|E_W|)$ prerequisite edges in PSP. PSP has a straightforward linear reduction to MAX-FLOW [8]. We use the Edmonds-Karp algorithm [42] for MAX-FLOW, which runs in time $\mathcal{O}(|N_W| \cdot |E_W|^2)$.

Impact of change detection precision and recall. The optimality of our algorithm for OEP assumes that the changes between iteration t and $t + 1$ have been identified perfectly. In reality, this maybe not be the case due to the intractability of change detection, as discussed in Section 4.2. An undetected change is a false negative in this case, while falsely identifying an unchanged operator as deprecated is a false positive. A detection mechanism with high precision lowers the chance of unnecessary recomputation, whereas anything lower than perfect recall leads to incorrect results. In our current approach, we opt for a detection mechanism that guarantee correctness under mild assumptions, at the cost of some false positives such as $a + b \neq b + a$.

5.3 OPTIMAL MATERIALIZATION PLAN

The OPT-MAT-PLAN (OMP) problem is tackled by HELIX’s materialization optimizer: while running workflow W_t at iteration t , intermediate results are selectively materialized for the purpose of accelerating execution in iterations $> t$. We now formally introduce OMP and show that it is NP-HARD even under strong assumptions. We propose an online heuristic for OMP that runs in linear time and achieves good reuse rate in practice (as we will show in Chapter 6), in addition to minimizing memory footprint by avoiding unnecessary caching of intermediate results.

Materialization cost. We let s_i denote the *storage cost* for materializing n_i , representing the size of n_i on disk. When loading n_i back from disk to memory, we have the following relationship between load time and storage cost: $l_i = s_i / (\text{disk read speed})$. For simplicity, we also assume the time to write n_i to disk is the same as the time for loading it from disk, i.e., l_i . We can easily generalize to the setting where load and write latencies are different.

To quantify the benefit of materializing intermediate results at iteration t on subsequent iterations, we formulate the *materialization run time* $T_M(W_t)$ to capture the tradeoff between the additional time to materialize intermediate results and the run time reduction in iteration $t + 1$. Although materialized results can be reused in multiple future iterations, we only consider the $(t + 1)$ th iteration since the total number of future iterations, \mathcal{T} , is unknown.

Since modeling \mathcal{T} is a complex open problem, we defer the amortization model to future work.

Definition 5.1 *Given a workflow W_t , operator metrics c_i, l_i, s_i for every $n_i \in N_t$, and a subset of nodes $M \subseteq N_t$, the materialization run time is defined as*

$$T_M(W_t) = \sum_{n_i \in M} l_i + T^*(W_{t+1}) \quad (5.8)$$

where $\sum_{n_i \in M} l_i$ is the time to materialize all nodes selected for materialization, and $T^*(W_t)$ is the optimal workflow run time obtained using the algorithm in Section 5.2, with M materialized.

Equation 5.8 defines the optimization objective for OMP.

Problem 5.3 (OPT-MAT-PLAN) *Given a Workflow W_t with DAG $G_W^t = (N_t, E_t)$ at iteration t and a storage budget S , find a subset of nodes $M \subseteq N_t$ to materialize at t in order to minimize $T_M(W_t)$, while satisfying the storage constraint $\sum_{n_i \in M} s_i \leq S$.*

Let M^* be the optimal solution to OMP, i.e.,

$$\operatorname{argmin}_{M \subseteq N_t} \sum_{n_i \in M} l_i + T^*(W_{t+1}) \quad (5.9)$$

As discussed in [18], there are many possibilities for W_{t+1} , and they vary by application domain. User modeling and predictive analysis of W_{t+1} itself is a substantial research topic that we will address in future work. This user model can be incorporated into OMP by using the predicted changes to better estimate the likelihood of reuse for each operator. However, even under very restrictive assumptions about W_{t+1} , we can show that OPT-MAT-PLAN is NP-HARD, via a simple reduction from the KNAPSACK problem.

Theorem 5.2 OPT-MAT-PLAN is NP-hard.

We show that OMP is NP-hard under restrictive assumptions about the structure of W_{t+1} relative to W_t , which implies the general version of OMP is also NP-hard.

In our proof we make the simplifying assumption that all nodes in the Workflow DAG are reusable in the next iteration:

$$n_i^t \equiv n_i^{t+1} \quad \forall n_i^t \in N_t, n_i^{t+1} \in N_{t+1} \quad (5.10)$$

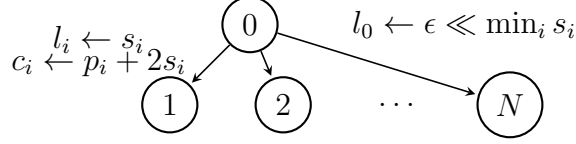


Figure 5.2: OMP DAG for Knapsack reduction.

Under this assumption, we achieve maximum reusability of materialized intermediate results since all operators that persist across iterations t and $t + 1$ are equivalent. We use this assumption to sidestep the problem of predicting iterative modifications, which is a major open problem by itself.

Our proof for the NP-hardness of OMP subject to Eq(5.10) uses a reduction from the known NP-hard Knapsack problem.

Problem 5.4 (*Knapsack*) Given a knapsack capacity B and a set N of n items, with each $i \in N$ having a size s_i and a profit p_i , find $S^* =$

$$\operatorname{argmax}_{S \subseteq N} \sum_{i \in S} p_i \quad (5.11)$$

such that $\sum_{i \in S^*} s_i \leq B$.

For an instance of Knapsack, we construct a simple Workflow DAG W as shown in Figure 5.2. For each item i in Knapsack, we construct an output node n_i with $l_i = s_i$ and $c_i = p_i + 2s_i$. We add an input node n_0 with $l_0 = \epsilon < \min s_i$ that all output nodes depend on. Let $Y_i \in \{0, 1\}$ indicate whether a node $n_i \in M$ in the optimal solution to OMP in Eq (5.9) and $X_i \in \{0, 1\}$ indicate whether an item is picked in the Knapsack problem. We use B as the storage budget, i.e., $\sum_{i \in \{0, 1\}} Y_i l_i \leq B$.

Theorem 5.3 We obtain an optimal solution to the Knapsack problem for $X_i = Y_i \ \forall i \in \{1, 2, \dots, n\}$.

First, we observe that for each n_i , $T^*(W)$ will pick $\min(l_i, c_i)$ given the flat structure of the DAG. By construction, $\min(l_i, c_i) = l_i$ in our reduction. Second, materializing n_i helps in the first iteration only when it is loaded in the second iteration. Thus, we can rewrite Eq (5.9) as

$$\operatorname{argmin}_{\mathbf{Y} \in \{0, 1\}^N} \sum_{i=1}^N Y_i l_i + \left(\sum_{i=1}^N Y_i l_i + (1 - Y_i) c_i \right) \quad (5.12)$$

where $\mathbf{Y} = (Y_1, Y_2, \dots, Y_N)$. Substituting in our choices of l_i and c_i in terms of p_i and s_i in (5.12), we obtain $\operatorname{argmin}_{\mathbf{Y} \in \{0, 1\}^N} \sum_{i=1}^N -Y_i p_i$. Clearly, satisfying the storage constraint also

Algorithm 5.2: Streaming OMP

Data: $G_w = (N, E), \{l_i\}, \{c_i\}, \{s_i\}$, storage budget S

```
1  $M \leftarrow \emptyset$ ;  
2 while Workflow is running do  
3    $O \leftarrow \text{FindOutOfScope}(N)$ ;  
4   for  $n_i \in O$  do  
5     if  $C(n_i) > 2l_i$  and  $S - s_i \geq 0$  then  
6       Materialize  $n_i$ ;  
7        $M \leftarrow M \cup \{n_i\}$ ;  
8        $S \leftarrow S - s_i$ 
```

satisfies the budget constraint in Knapsack by construction. Thus, the optimal solution to OMP as constructed gives the optimal solution to Knapsack.

Streaming constraint. Even when W_{t+1} is known, solving OPT-MAT-PLAN optimally requires knowing the run time statistics for all operators, which can be fully obtained only at the end of the workflow. Deferring materialization decisions until the end requires all intermediate results to be cached or recomputed, which imposes undue pressure on memory and cripples performance. Unfortunately, reusing statistics from past iterations as in Section 5.2 is not viable here because of the cold-start problem—materialization decisions need to be made for new operators based on realistic statistics. Thus, to avoid slowing down execution with high memory usage, we impose the following constraint.

Definition 5.2 *Given a Workflow DAG $G_w = (N, E)$, $n_i \in N$ is out-of-scope at runtime if all children of n_i have been computed or reloaded from disk, thus removing all dependencies on n_i .*

Constraint 5.2 *Once n_i becomes out-of-scope, it is either materialized immediately or removed from cache.*

OMP Heuristics. We now describe the heuristic employed by HELIX to approximate OMP while satisfying Constraint 5.2.

Definition 5.3 *Given Workflow DAG $G_w = (N, E)$, the cumulative run time for a node n_i is defined as*

$$C(n_i) = t(n_i) + \sum_{n_j \in \text{ancestors}(n_i)} t(n_j) \quad (5.13)$$

where $t(n_i) = \mathbb{I}\{s(n_i) = S_c\} c_i + \mathbb{I}\{s(n_i) = S_l\} l_i$.

Algorithm 5.2 shows the heuristics employed by HELIX’s materialization optimizer to decide what intermediate results to materialize. In essence, Algorithm 5.2 decides to materialize if twice the load cost is less than the cumulative run time for a node. The intuition behind this algorithm is that assuming loading a node allows all of its ancestors to be pruned, the materialization time in iteration t and the load time in iteration $t + 1$ combined should be less than the total pruned compute time, for the materialization to be cost effective.

Note that the decision to materialize does not depend on which ancestor nodes have been previously materialized. The advantage of this approach is that regardless of where in the workflow the changes are made, the reusable portions leading up to the changes are likely to have an efficient execution plan. That is to say, if it is cheaper to load a reusable node n_i than to recompute, Algorithm 5.2 would have materialized n_i previously, allowing us to make the right choice for n_i . Otherwise, Algorithm 5.2 would have materialized some ancestor n_j of n_i such that loading n_j and computing everything leading to n_i is still cheaper than loading n_i .

Due to the streaming Constraint 5.2, complex dependencies between descendants of ancestors such as the one between n_5 and n_8 in Figure 5.1 previously described in Section 5.2, are ignored by Algorithm 5.2—we cannot retroactively update our decision for n_5 after n_8 has been run. We show in Chapter 6 that this simple algorithm is effective in multiple application domains.

Limitations of Streaming OMP. The streaming OMP heuristic given in Algorithm 5.2 can behave poorly in pathological cases. For one simple example, consider a workflow given by a chain DAG of m nodes, where node n_i (starting from $i = 1$) is a prerequisite for node n_{i+1} . If node n_i has $l_i = i$ and $c_i = 3$, for all i , then Algorithm 5.2 will choose to materialize every node, which has storage costs of $\mathcal{O}(m^2)$, whereas a smarter approach would only materialize later nodes and perhaps have storage cost $\mathcal{O}(m)$. If storage is exhausted because Algorithm 5.2 persists too much early on, this could easily lead to poor execution times in later iterations. We did not observe this sort of pathological behavior in our experiments.

Mini-Batches. In the stream processing (to be distinguished from the streaming constraint in Constraint 5.2) where the input is divided into mini batches processed end-to-end independently, Algorithm 5.2 can be adapted as follows: 1) make materialization decisions using the load and compute time for the first mini batch processed end-to-end; 2) reuse the same decisions for all subsequent mini batches for each operator. This approach avoids dataset fragmentation that complicates reuse for different workflow versions. We plan on investigating other approaches for adapting HELIX for stream processing in future work.

5.4 WORKFLOW DAG PRUNING

In addition to optimizations involving intermediate result reuse, HELIX further reduces overall workflow execution time by time by pruning extraneous operators from the **Workflow DAG**.

HELIX performs pruning by applying program slicing on the **Workflow DAG**. In a nutshell, HELIX traverses the DAG backwards from the output nodes and prunes away any nodes not visited in this traversal. Users can explicitly guide this process in the programming interface through the `has_extractors` and `uses` keywords, described in Table 3.2. An example of an Extractor pruned in this fashion is `raceExt`(grayed out) in Figure 2.4b), as it is excluded from the `rows has_extractors` statement. This allows users to conveniently perform manual feature selection using domain knowledge.

HELIX provides two additional mechanisms for pruning operators other than using the lack of output dependency, described next.

Data-Driven Pruning. Furthermore, HELIX inspects relevant data to automatically identify operators to prune. The key challenge in *data-driven pruning* is data lineage tracking across the entire workflow. For many existing systems, it is difficult to trace features in the learned model back to the operators that produced them. To overcome this limitation, HELIX performs additional provenance bookkeeping to track the operators that led to each feature in the model when converting DPR output to ML-compatible formats. An example of data-driven workflow optimization enabled by this bookkeeping is pruning features by model weights. Operators resulting in features with zero weights can be pruned without changing the prediction outcome, thus lowering the overall run time without compromising model performance.

Data-driven pruning is a powerful technique that can be extended to unlock the possibilities for many more impactful automatic workflow optimizations. Possible future work includes using this technique to minimize online inference time in large scale, high query-per-second settings and to adapt the workflow online in stream processing.

Cache Pruning. While Spark, the underlying data processing engine for HELIX, provides automatic data uncaching via a least-recently-used (LRU) scheme, HELIX improves upon the performance by actively managing the set of data to evict from cache. From the DAG, HELIX can detect when a node becomes out-of-scope. Once an operator has finished running, HELIX analyzes the DAG to uncach newly out-of-scope nodes. Combined with the lazy evaluation order, the intermediate results for an operator reside in cache only when it is immediately needed for a dependent operator.

One limitation of this eager eviction scheme is that any dependencies undetected by HELIX,

such as the ones created in a UDF, can lead to premature uncaching of DCs before they are truly out-of-scope. The `uses` keyword in HML, described in Table 3.2, provides a mechanism for users to manually prevent this by explicitly declaring a UDF’s dependencies on other operators. In the future, we plan on providing automatic UDF dependency detection via introspection.

CHAPTER 6: EMPIRICAL EVALUATION

The goal of our evaluation is to test if HELIX 1) *supports* ML workflows in a variety of application domains; 2) *accelerates* iterative execution through intermediate result reuse, compared to other ML systems that don’t optimize iteration; 3) is *efficient*, enabling optimal reuse without incurring a large storage overhead.

6.1 SYSTEMS AND BASELINES FOR COMPARISON

We compare the optimized version of HELIX, HELIX OPT, against two state-of-the-art ML workflow systems: KeystoneML [6], and DeepDive [7]. In addition, we compare HELIX OPT with two simpler versions, HELIX AM and HELIX NM. While we compare against DeepDive, and KeystoneML to verify 1) and 2) above, HELIX AM and HELIX NM are used to verify 3). We describe each of these variants below:

KeystoneML. KeystoneML [6] is a system, written in Scala and built on top of Spark, for the construction of large scale, end-to-end, ML pipelines. KeystoneML specializes in classification tasks on structured input data. No intermediate results are materialized in KeystoneML, as it does not optimize execution across iterations.

DeepDive. DeepDive [7, 44] is a system, written using Bash scripts and Scala for the main engine, with a database backend, for the construction of end-to-end information extraction pipelines. Additionally, DeepDive provides limited support for classification tasks. All intermediate results are materialized in DeepDive.

HELIX Opt. A version of HELIX that uses Algorithm 5.1 for the optimal reuse strategy and Algorithm 5.2 to decide what to materialize.

HELIX AM. A version of HELIX that uses the same reuse strategy as HELIX OPT and *always*

	Census ([43])	Genomics ([3])	IE ([44])	MNIST ([6])
Num. Data Source	Single	Multiple	Multiple	Single
Input to Example	One-to-One	One-to-Many	One-to-Many	One-to-One
Feature Granularity	Fine Grained	N/A	Fine Grained	Coarse Grained
Learning Task Type	Supervised; Class.	Unsupervised	Structured Prediction	Supervised; Class.
Application Domain	Social Sciences	Natural Sciences	NLP	Computer Vision
HELIX	✓	✓	✓	✓
KeystoneML	✓	✓		✓*
DeepDive	✓*		✓*	

Table 6.1: Summary of workflow characteristics and support by the systems compared. Grayed out cells indicate that the system in the row does not support the workflow in the column. ✓* indicates that the implementation is by the original developers of DeepDive/KeystoneML. “Class.” is short for classification.

materializes all intermediate results.

HELIX NM. A version of HELIX that uses the same reuse strategy as HELIX OPT and *never materializes* any intermediate results.

6.2 WORKFLOWS

We conduct our experiments using four real-world ML workflows spanning a range of application domains. Table 6.1 summarizes the characteristics of the four workflows, described next. We are interested in four properties when characterizing each workflow:

- *Number of data sources:* whether the input data comes from a single source (e.g., a CSV file) or multiple sources (e.g., documents and a knowledge base), necessitating joins.
- *Input to example mapping:* the mapping from each input data unit (e.g., a line in a file) to each learning example for ML. One-to-many mappings require more complex data preprocessing than one-to-one mappings.
- *Feature granularity:* fine-grained features involve applying extraction logic on a specific piece of the data (e.g., 2nd column) and are often application-specific, whereas coarse-grained features are obtained by applying an operation, usually a standard DPR technique such as normalization, on the entire dataset.
- *Learning task type:* while classification and structured prediction tasks both fall under supervised learning for having observed labels, structured prediction workflows involve more complex data preprocessing and models; unsupervised learning tasks do not have known labels, so they often require more qualitative and fine-grained analyses of outputs.

Census Workflow. This workflow corresponds to a classification task with simple features from structured inputs from the DeepDive Github repository [43]. It uses the Census Income dataset [45], with 14 attributes representing demographic information, with the goal to predict whether a person’s annual income is $\geq 50K$, using fine-grained features derived from input attributes. The complexity of this workflow is representative of use cases in the social and natural sciences, where covariate analysis is conducted on well-defined variables. HELIX code for the initial version of this workflow is shown in Figure 2.4a). This workflow evaluates a system’s efficiency in handling simple ML tasks with fine-grained feature engineering.

Genomics Workflow. This workflow is described in Example 1.1, involving two major steps: 1) split the input articles into words and learn vector representations for entities of interest, identified by joining with a genomic knowledge base, using word2vec [4]; 2) cluster the vector representation of genes using K-Means to identify functional similarity. Each input record is an article, and it maps onto many gene names, which are training examples. This workflow has minimal data preprocessing with no specific features but involves multiple learning steps. Both learning steps are unsupervised, which leads to more qualitative and exploratory evaluations of the model outputs than the standard metrics used for supervised learning. We include a workflow with unsupervised learning and multiple learning steps to verify that the system is able to accommodate variability in the learning task.

Information Extraction (IE) Workflow. This workflow involves identifying mentions of spouse pairs from news articles, using a knowledge-base of known spouse pairs, from DeepDive [44]. The objective is to extract structured information from unstructured input text, using complex fine-grained features such as part-of-speech tagging. Each input article contains ≥ 0 spouse pairs, hence creating a one-to-many relationship between input records and learning examples. This workflow exemplifies use cases in information extraction, and tests a system’s ability to handle joins and complex data preprocessing.

MNIST Workflow. The MNIST dataset [46] contains images of handwritten digits to be classified, which is a well-studied task in the computer vision community, from the KeystoneML [6] evaluation. The workflow involves nondeterministic (and hence not reusable) data preprocessing, with a substantial fraction of the overall run time spent on L/I in a typical iteration. We include this application to ensure that in the extreme case where there is little reuse across iterations, HELIX does not incur a large overhead.

Each workflow was implemented in HELIX, and if supported, in DeepDive and KeystoneML, with \checkmark^* in Table 6.1 indicating that we used an existing implementation by the developers of DeepDive or KeystoneML, which can be found at:

- Census DeepDive: <https://github.com/HazyResearch/deepdive/blob/master/examples/census/app.ddlog>
- IE DeepDive: <https://github.com/HazyResearch/deepdive/blob/master/examples/spouse/app.ddlog>
- MNIST KeystoneML: <https://github.com/amplab/keystone/blob/master/src/main/scala/keystoneml/pipelines/images/mnist/MnistRandomFFT.scala>

DeepDive has its own DSL, while KeystoneML’s programming interface is an embedded DSL in Scala, similar to HML. We explain limitations that prevent DeepDive and KeystoneML

from supporting certain workflows (grey cells) in Section 6.5.1.

6.3 RUNNING EXPERIMENTS

Simulating iterative development. In our experiments, we modify the workflows to simulate typical iterative development by a ML application developer or data scientist. Instead of arbitrarily choosing operators to modify in each iteration, we use the iteration frequency in Figure 3 from our literature study [18] to determine the type of modifications to make in each iteration, for the specific domain of each workflow. We convert the iteration counts into fractions that represent the likelihood of a certain type of change. At each iteration, we draw an iteration type from {DPR, L/I, PPR} according to these likelihoods. Then, we randomly choose an operator of the drawn type and modify its source code. For example, if an “L/I” iteration were drawn, we might change the regularization parameter for the ML model. We run 10 iterations per workflow (except NLP, which has only DPR iterations), double the average iteration count found in our survey in Section 2.2.

Note that in real world use, the modifications in each iteration are entirely up to the user. HELIX is not designed to suggest modifications, and the modifications chosen in our experiments are for evaluating only system run time and storage use. We use statistics aggregated over > 100 papers to determine the iterative modifications in order to simulate behaviors of the *average domain expert* more realistically than arbitrary choice.

Environment. All single-node experiments are run on a server with 125 GiB of RAM, 16 cores on 8 CPUs (Intel Xeon @ 2.40GHz), and 2TB HDD with 170MB/s as both the read and write speeds. Distributed experiments are run on nodes each with 64GB of RAM, 16 cores on 8 CPUs (Intel Xeon @ 2.40GHz), and 500GB of HDD with 180MB/s as both the read and write speeds. We set the storage budget in HELIX to 10GB. That is, 10GB is the maximum accumulated disk storage for HELIX OPT at all times during the experiments. After running the initial version to obtain the run time for iteration 0, a workflow is modified according to the type of change determined as above. In all four systems the modified workflow is recompiled. In DeepDive, we rerun the workflow using the command `deepdive run`. In HELIX and KeystoneML, we resubmit a job to Spark in local mode. We use Postgres as the database backend for DeepDive. Although HELIX and KeystoneML support distributed execution via Spark, DeepDive needs to run on a single server. Thus, we compare against all systems on a single node and additionally compare against KeystoneML on clusters.

6.4 METRICS

We evaluate each system’s ability to support diverse ML tasks by qualitative characterization of the workflows and use-cases supported by each system. Our primary metric for workflow execution is *cumulative run time* over multiple iterations. The cumulative run time considers only the run time of the workflows, not any human development time. We measure with wall-clock time because it is the latency experienced by the user. When computing cumulative run times, we average the per-iteration run times over five complete runs for stability. Note that the per-iteration time measures both the time to execute the workflow and any time spent to materialize intermediate results. We also measure *memory usage* to analyze the effect of batch processing, and measure *storage size* to compare the run time reduction to storage ratio of time-efficient approaches. Storage is compared only for variants of HELIX since other systems do not support automatic reuse.

6.5 EVALUATION VS. STATE-OF-THE-ART SYSTEMS

6.5.1 Use Case Support

Recall that the four workflows used in our experiments are in social sciences, NLP, computer vision, and natural sciences, respectively. Table 6.1 lists the characteristics of each workflow and the three systems’ ability to support it. Both KeystoneML and DeepDive have limitations that prevent them from supporting certain types of tasks. The pipeline programming model in KeystoneML is effective for large scale classification and can be adapted to support unsupervised learning. However, it makes fine-grained features cumbersome to program and is not conducive to structured prediction tasks due to complex data preprocessing. On the other hand, DeepDive is highly specialized for information extraction and focuses on supporting data preprocessing. Unfortunately, its learning and evaluation components are not configurable by the user, limiting the type of ML tasks supported. DeepDive is therefore unable to support the MNIST and genomics workflows, both of which required custom ML models. Additionally, we are only able to show DeepDive performance for DPR iterations for the supported workflows in our experiments.

6.5.2 Cumulative Run Time

Figure 6.1 shows the cumulative run time for all four workflows. The x-axis shows the iteration number, while the y-axis shows the cumulative run time in log scale at the i th

iteration. Each point represents the cumulative run time of the first i iterations. The color under the curve indicates the workflow component modified in each iteration (purple = DPR, orange = L/I, green = PPR). For example, the DPR component was modified in the first iteration of Census. Figure 6.2 shows the breakdown by workflow components and materialization for the individual iteration run times in HELIX, with the same color scheme as in Figure 6.1 for the workflow components and gray for materialization time.

Census. As shown in Figure 6.1(a), the census workflow has the largest cumulative run time gap between HELIX OPT and the competitor systems—HELIX OPT *is 19 \times faster than KeystoneML as measured by cumulative run time over 10 iterations*. By materializing and reusing intermediate results HELIX OPT is able to substantially reduce cumulative run-time relative to other systems. Figure 6.2(a) shows that 1) on PPR iterations HELIX recomputes only the PPR; 2) the materialization of L/I outputs, which allows the pruning of DPR and L/I in PPR iterations, takes considerably less time than the compute time for DPR and L/I; 3) HELIX OPT reruns DPR in iteration 5 (L/I) because HELIX OPT avoided materializing the large DPR output in a previous iteration. For the first three iterations, which are DPR (the only type of iterations DeepDive supports), the 2 \times reduction between HELIX OPT and DeepDive is due to the fact that DeepDive does data preprocessing with Python and shell scripts, while HELIX OPT uses Spark. While both KeystoneML and HELIX OPT use Spark, KeystoneML takes longer on DPR and L/I iterations than HELIX OPT due to a longer L/I time incurred by its caching optimizer’s failing to cache the training data for learning. The dominant number of PPR iterations for this workflow reflects the fact that users in the social sciences conduct extensive fine-grained analysis of results, per our literature survey [18].

Genomics. In Figure 6.1(b), HELIX OPT shows a 3 \times speedup over KeystoneML for the genomics workflow. The materialize-nothing strategy in KeystoneML clearly leads to no run time reduction in subsequent iterations. HELIX OPT, on the other hand, shows a per-iteration run time that is proportional to the number of operators affected by the change in that iteration. Figure 6.2(b) shows that 1) in PPR iterations HELIX OPT has near-zero run time, enabled by a small materialization time in the prior iteration; 2) one of the ML models takes considerably more time, and HELIX OPT is able to prune it in iteration 4 since it is not changed.

NLP. Figure 6.1(c) shows that the cumulative run time for both DeepDive and HELIX OPT increases linearly with iteration for the NLP workflow, but at a much higher rate for DeepDive than HELIX OPT. This is due to the lack of automatic reuse in DeepDive. The first operator in this workflow is a time-consuming NLP parsing operator, whose results are reusable for all subsequent iterations. While both DeepDive and HELIX OPT materialize this operator in the first iteration, DeepDive does not automatically reuse the results. HELIX

OPT, on the other hand, consistently prunes this NLP operation in all subsequent iterations, as shown in Figure 6.2(c), leading to large run time reductions in iterations 1-5 and thus a large cumulative run time reduction.

MNIST. Figure 6.1(d) shows the cumulative run times for the MNIST workflow. As mentioned above, the MNIST workflow has nondeterministic data preprocessing, which means any changes to the DPR and L/I components prevents safe reuse of any intermediate result. However, iterations containing only PPR changes can reuse intermediates for DPR and L/I had they been materialized previously. Furthermore, we found that the DPR run time is short but cumulative size of all DPR intermediates is large. Thus, materializing all these DPR intermediates would incur a large run time overhead. KeystoneML, which does not materialize any intermediate results, shows a linear increase in cumulative run time due to no reuse. HELIX OPT, on the other hand, only shows slight increase in runtime over KeystoneML for DPR and L/I iterations because it is only materializing the L/I results on these iterations, not the nonreusable, large DPR intermediates. In Figure 6.2(d), we see 1) DPR operations take negligible time, and HELIX OPT avoids wasteful materialization of their outputs; 2) the materialization time taken in the DPR and L/I iterations pays off for HELIX OPT in PPR iterations, which take negligible run time due to reuse.

6.5.3 Scalability vs. KeystoneML

Dataset Size. We test scalability of HELIX and KeystoneML with respect to dataset size by running the ten iterations in Figure 6.1(a) of the Census Workflow on two different sizes of the input. Census 10x is obtained by replicating Census ten times in order to preserve the learning objective. Figure 6.3(a) shows run time performance of HELIX and KeystoneML on the two datasets on a single node. Both yield 10x speedup over the smaller dataset, scaling linearly with input data size, but HELIX continues to dominate KeystoneML.

Cluster. We test scalability of HELIX and KeystoneML with respect to cluster size by running the same ten iterations in Figure 6.1(a) on Census 10x described above. Using a uniform set of machines, we create clusters with 2, 4, and 8 workers and run HELIX and KeystoneML on each of these clusters to collect cumulative run time.

Figure 6.3(b) shows that 1) HELIX has lower cumulative run time than KeystoneML on the same cluster size, consistent with the single node results; 2) KeystoneML achieves $\approx 45\%$ run time reduction when the number of workers is doubled, scaling roughly linearly with the number of workers; 3) From 2 to 4 workers, HELIX achieves up to 75% run time reduction 4) From 4 to 8 workers, HELIX sees a slight increase in run time. Recall from Chapter 3 that the semantic unit data structure in HML allows multiple transformer operations (e.g.,

indexing, computing discretization boundaries) to be learned using a single pass over the data via loop fusion. This reduces the communication overhead in the cluster setting, hence the super linear speedup in 3). On the other hand, the communication overhead for PPR operations outweighs the benefits of distributed computing, hence the slight increase in 4).

6.6 EVALUATION VS. SIMPLER HELIX VERSIONS

Next, we evaluate the effectiveness of Algorithm 5.2 at approximating the solution to the NP-hard OPT-MAT-PLAN problem. We compare HELIX OPT that runs Algorithm 5.2 against: HELIX AM that replaces Algorithm 5.2 with the policy to always materialize every operator, and HELIX NM that never materializes any operator. The two baseline heuristics present two performance extremes: HELIX AM maximizes storage usage, time for materialization, and the likelihood of being able to reuse unchanged results, whereas HELIX NM minimizes all three quantities. HELIX AM provides the most flexible choices for reuse. On the other hand, HELIX NM has no materialization time overhead but also offers no reuse.

Figures 6.5(a), (b), (e), and (f) show the cumulative run time on the same four workflows as in Figure 6.1 for the three variants.

HELIX AM is absent from Figures 6.5(e) and (f) because it did not complete within $50\times$ the time it took for other variants. The fact that HELIX AM failed to complete for the MNIST and NLP workflows demonstrate that indiscriminately materializing all intermediates can cripple performance. Figures 6.5(e) and (f) show that HELIX OPT achieves substantial run time reduction over HELIX NM using very little materialization time overhead (where the red line is above the yellow line).

For the census and genomics workflows where the materialization time is not prohibitive, Figures 6.5(a) and (b) show that in terms of cumulative run time, HELIX OPT outperforms HELIX AM, which attains the best reuse as explained above. We also compare the storage usage by HELIX AM and HELIX NM for these two workflows. Figures 6.5(c) and (d) show the storage size snapshot at the end of each iteration. The x-axis is the iteration numbers, and the y-axis is the amount of storage (in KB) in log scale. The storage use for HELIX NM is omitted from these plots because it is always zero.

We find that HELIX OPT outperforms HELIX AM while using less than half the storage used by HELIX AM for the census workflow in Figure 6.5(c) and $\frac{1}{30}$ the storage of HELIX AM for the genomics workflow in Figure 6.5(d). Storage is not monotonic because HELIX purges any previous materialization of original operators prior to execution, and these operators may not be chosen for materialization after execution, thus resulting in a decrease in storage.

Furthermore, to study the optimality of Algorithm 5.2, we compare the distribution of

nodes in the prune, reload, and compute states S_p, S_l, S_c between HELIX OPT and HELIX AM for workflows with HELIX AM completed in reasonable times. Since everything is materialized in HELIX AM, it achieves maximum reuse in the next iteration. Figure 6.4 shows that HELIX OPT enables the exact same reuse as HELIX AM, demonstrating its effectiveness on real workflows.

Overall, neither HELIX AM nor HELIX NM is the dominant strategy in all scenarios, and both can be suboptimal in some cases.

6.7 MEMORY USAGE BY HELIX

We evaluate memory usage by HELIX to ensure that its materialization and reuse benefits do not come at the expense of large memory overhead. We measure memory usage at one-second intervals during HELIX workflow execution. Figure 6.6 shows the peak and average memory used by HELIX in each iteration for all four workflows. We allocate 30GB memory (25% of total available memory) in the experiments. We observe that HELIX runs within the memory constraints on all workflows. Furthermore, on iterations where HELIX reuses intermediate results to achieve a high reduction in run time compared to other systems, memory usage is also significantly reduced. This indicates that HELIX reuses small intermediates that enable the pruning of a large portion of the subgraph to reduce run time, instead of overloading memory.

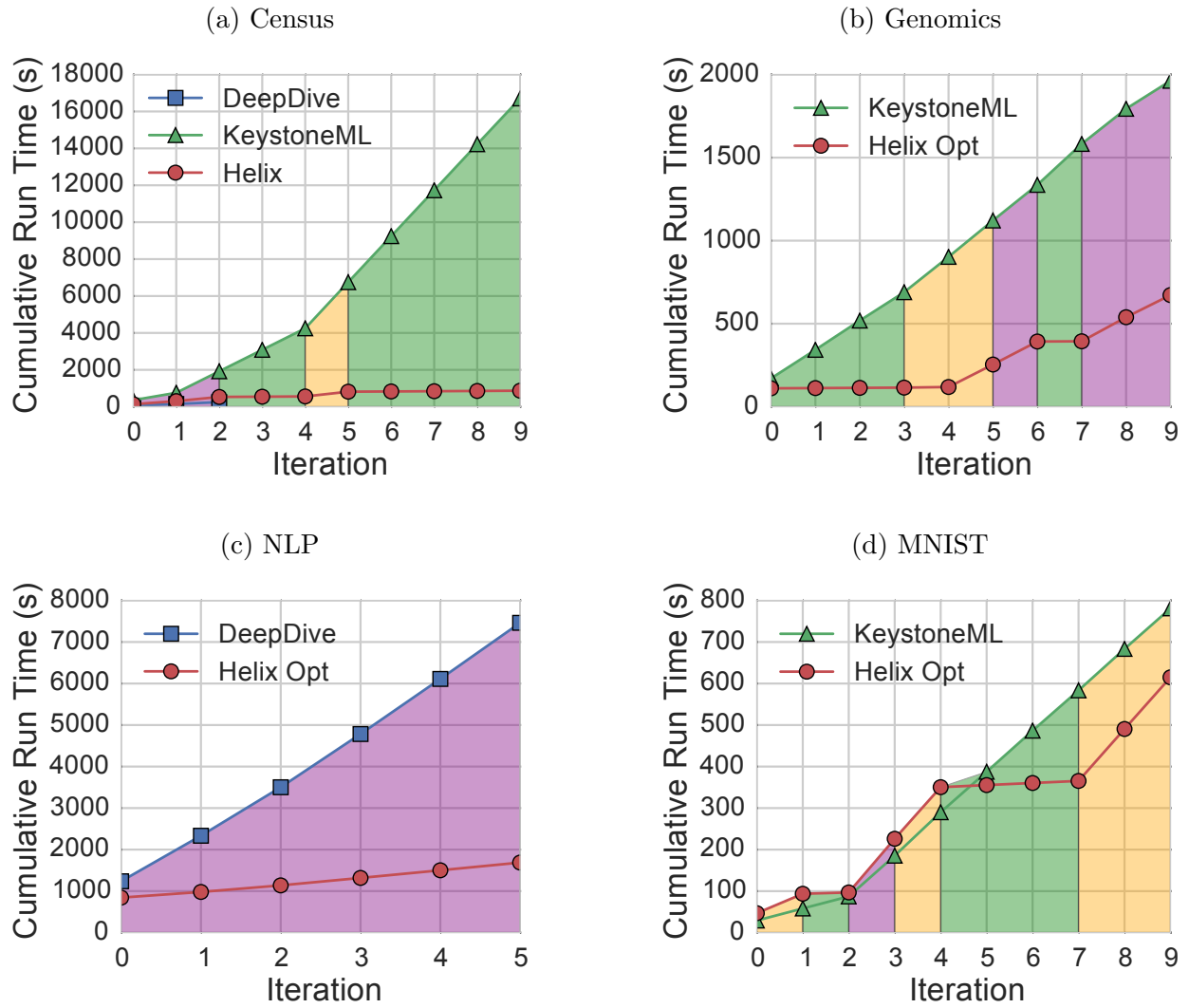


Figure 6.1: Cumulative run time for the four workflows. The color under the curve indicates the type of change in each iteration: purple for DPR, orange for L/I, and green for PPR.

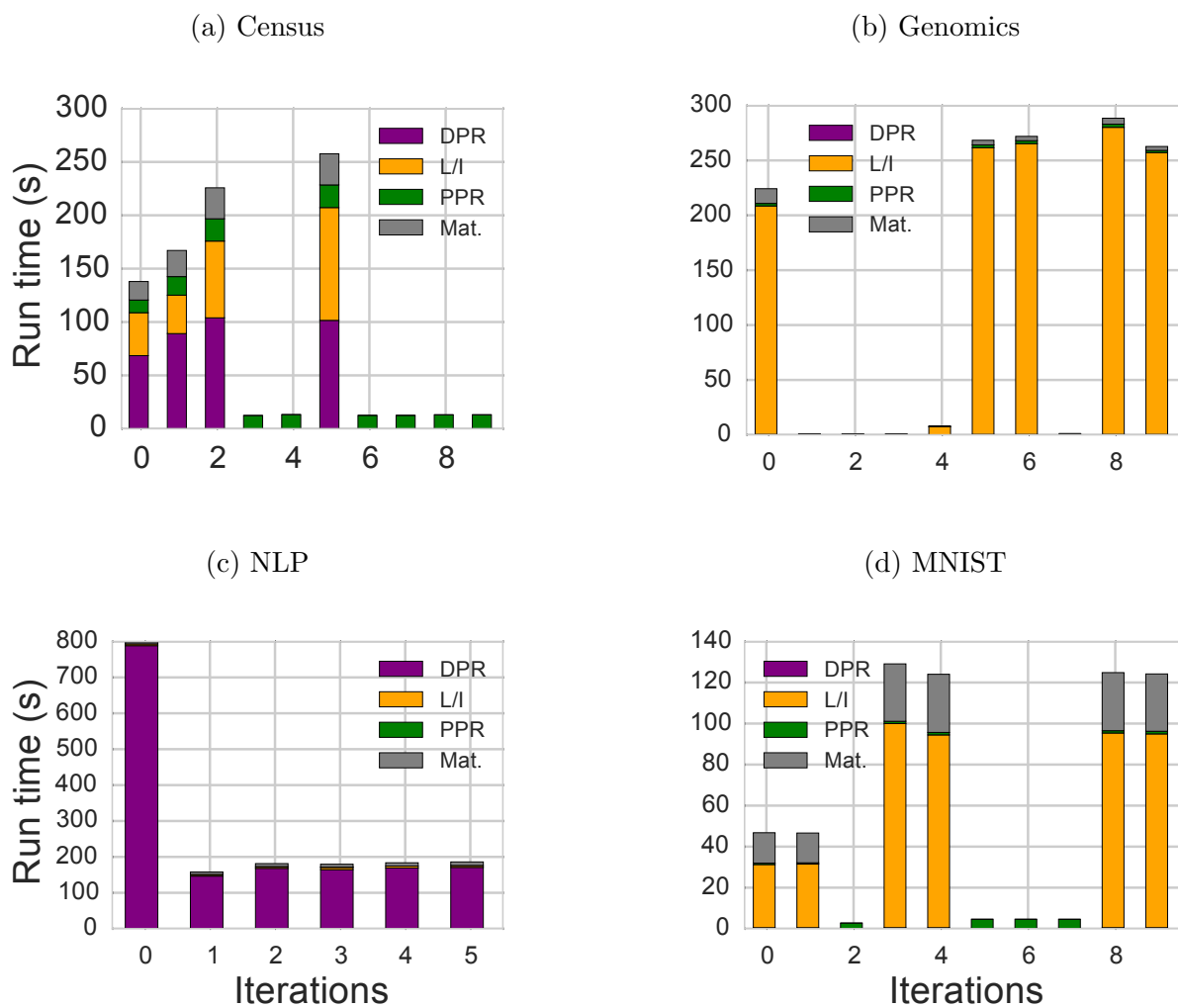


Figure 6.2: Run time breakdown by workflow component and materialization time per iteration for HELIX.

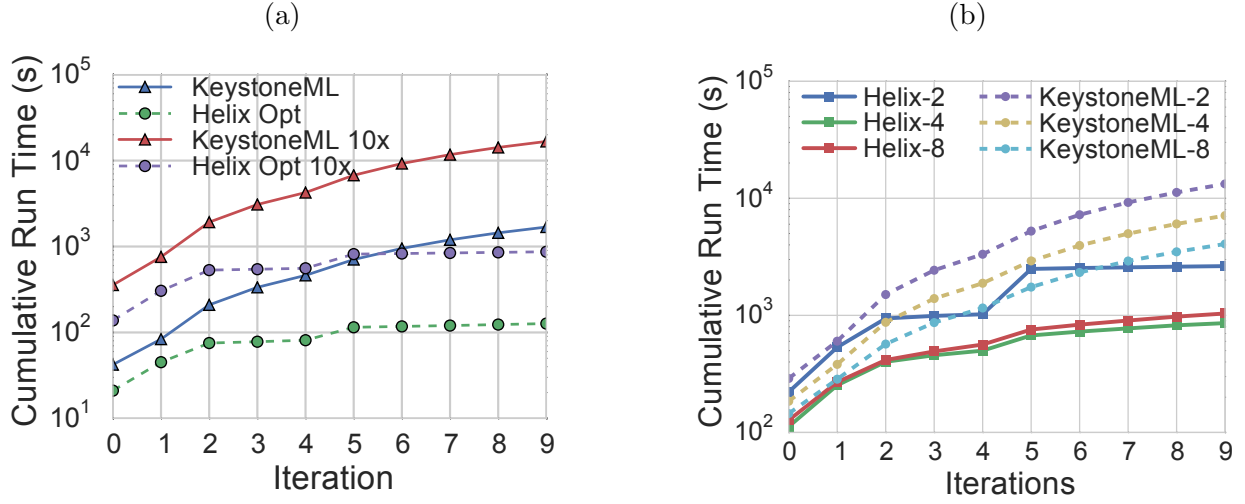


Figure 6.3: a) Census and Census 10x cumulative run time for HELIX and KeystoneML on a single node; b) Census 10x cumulative run time for HELIX and KeystoneML on different size clusters.

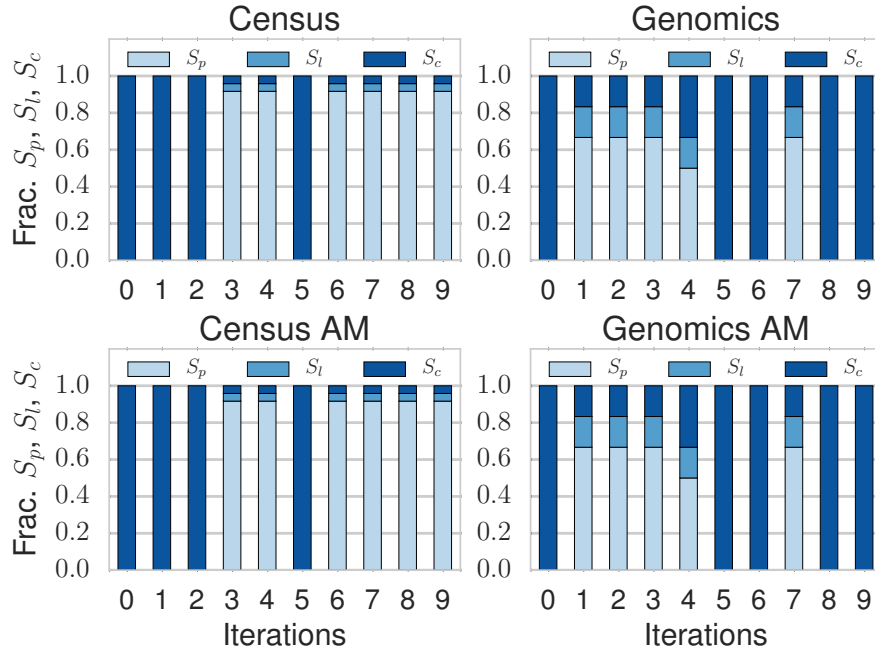


Figure 6.4: Fraction of states in S_p, S_l, S_c as determined by Algorithm 5.1 for the Census and Genomics workflows for HELIX OPT and HELIX AM.

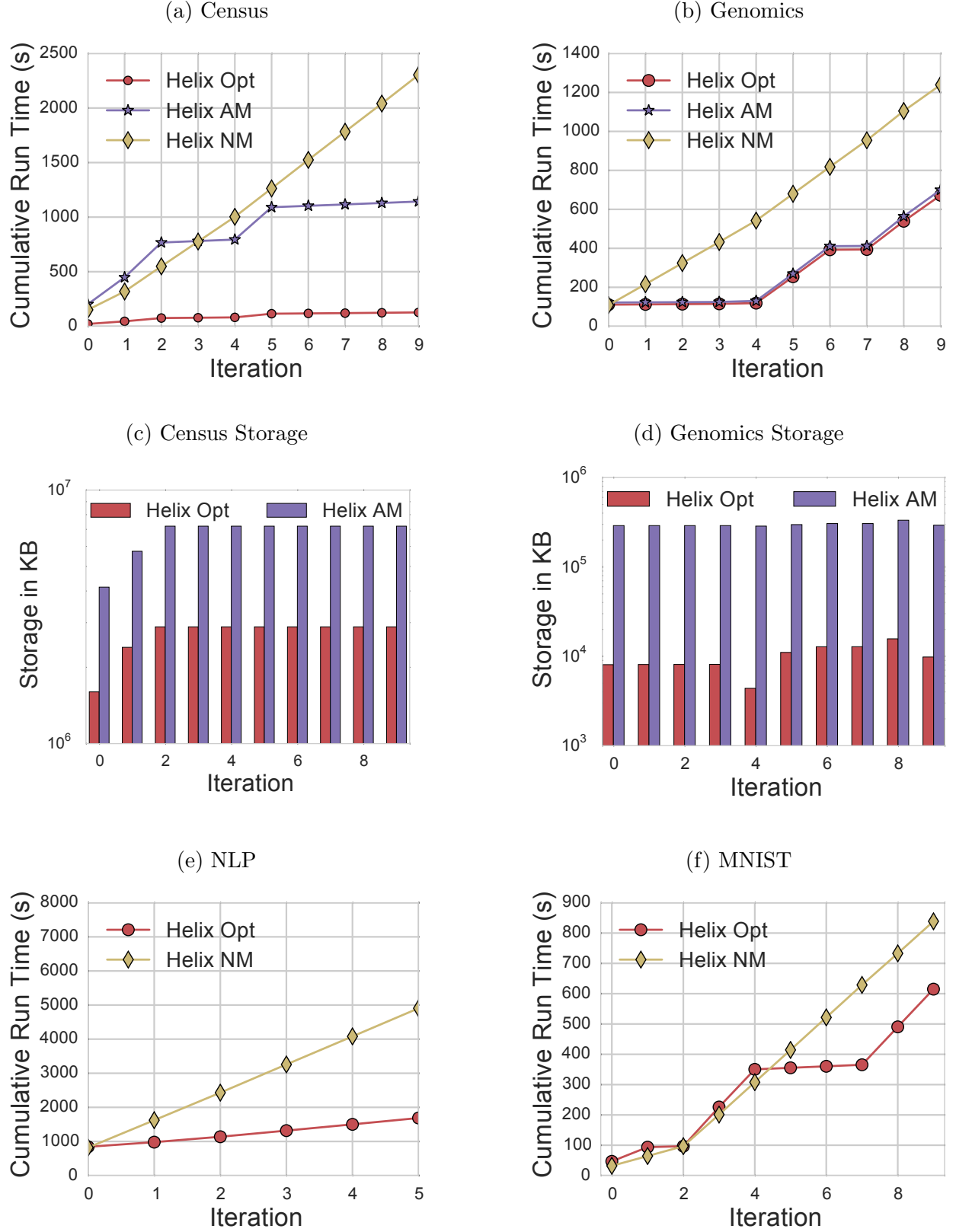


Figure 6.5: Cumulative run time and storage use against materialization heuristics on the same four workflows as in Figure 6.1.

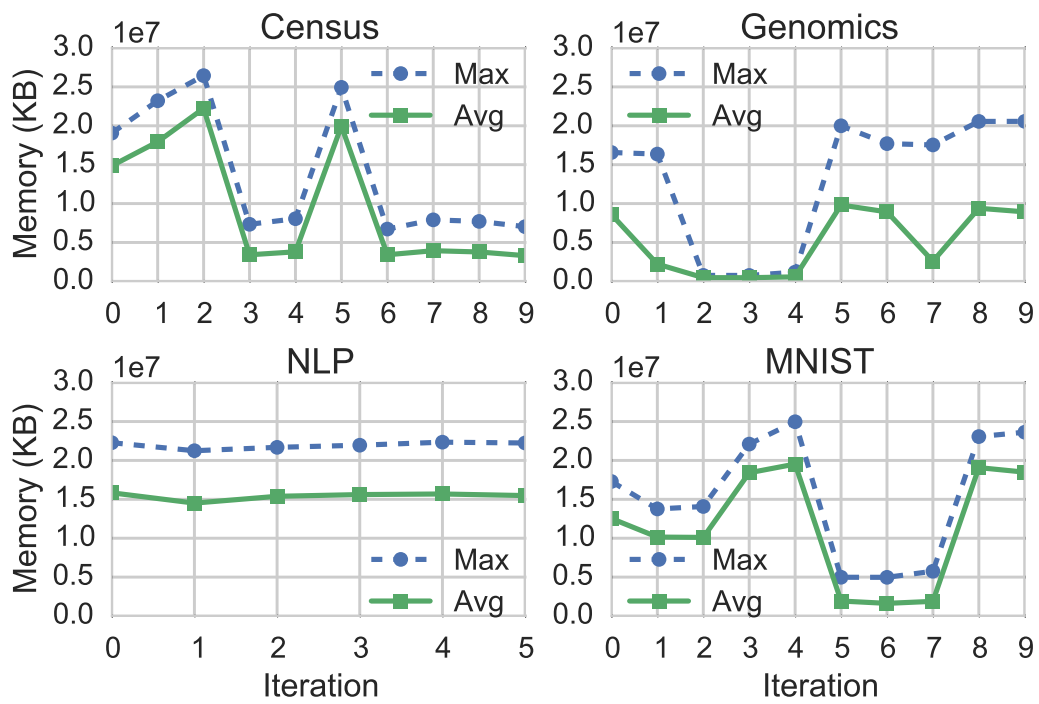


Figure 6.6: Peak and average memory for HELIX.

CHAPTER 7: RELATED WORK

Many systems have been developed in recent years to better support ML workflows. We begin by describing ML systems and other general workflow management tools, followed by systems that target the reuse of intermediate results.

Machine Learning Systems. We describe machine learning systems that support declarative programming, followed by other general-purpose systems that optimize across frameworks.

Declarative Systems. Due to the challenges in developing ML workflows, there has been recent efforts to make it easier to do so declaratively. Boehm et al. categorize declarative ML systems into three groups based on the usage: *declarative ML algorithms*, *ML libraries*, and *declarative ML tasks* [1]. Systems that support *declarative ML algorithms*, such as TensorFlow [31], SystemML [47], OptiML [48], ScalOps [49], and SciDB [50], allow ML experts to program new ML algorithms, by declaratively specifying linear algebra and statistical operations at higher levels of abstraction. Although it also builds a computation graph like HELIX, TensorFlow has no intermediate reuse and always performs a full computation e.g. any in-graph data preparation. TensorFlow’s lower level linear algebra operations are not conducive to data preprocessing. HELIX handles reuse at a higher level than TensorFlow ops. *ML libraries*, such as Mahout [51], Weka [30], GraphLab [52], Vowpal Wabbit [53], MLlib [26] and Scikit-learn [25], provide simple interfaces to optimized implementations of popular ML algorithms. TensorFlow has also recently started providing TFLearn [54], a high level ML library targeted at deep learning. Systems that support *declarative ML tasks* allow application developers with limited ML knowledge to develop models using higher-level primitives than in declarative ML algorithms. HELIX falls into this last group of systems, along with DeepDive [7, 44] and KeystoneML [6]. These systems perform workflow-level optimizations to reduce end-to-end execution time. Finally, at the extreme end of this spectrum are systems for in-RDBMS analytics [55, 56, 57] that extend databases to support ML, at great cost to flexibility.

Declarative ML task systems, like HELIX, can seamlessly make use of improvements in ML library implementations, such as MLlib [26], CoreNLP [20] and DeepLearning4j [10], within UDF calls. Unlike declarative ML algorithm systems, that are targeted at ML experts and researchers, these systems focus on end-users of ML.

Systems that Optimize Across Frameworks. These systems target a broad range of use-cases, including ML. Weld [58] and Tupleware [59] optimize UDFs written in different frameworks by compiling them down to a common intermediate representation. Declarative ML task

systems like HELIX can take advantage of the optimized UDF implementations; unlike HELIX, these systems do not benefit from seamless specification, execution, and end-to-end optimizations across workflow components that come from a unified programming model.

Systems for Optimizing Data Preprocessing. The database community has identified various opportunities for optimizing DPR. Several approaches identify as a key bottleneck in DPR and optimize it [60, 61, 62, 63]. Kumar et al. [60] optimizes generalized linear models directly over factorized / normalized representations of relational data, avoiding key-foreign key joins. Morpheus [61] and F [62] extend this factorized approach to general linear algebra operations and linear regression models, respectively (the latter over arbitrary joins). Some work [63] even attempts to characterize when joins can be eschewed altogether, without sacrificing performance. All of these optimizations are orthogonal to those used by HELIX. Another direction aims at reducing the manual effort involved in data cleaning and feature engineering [64, 65, 66, 67, 68]. All of these optimizations are orthogonal to those used by HELIX, which targets end-to-end iterative optimizations. Snorkel [64] supports training data engineering using rules. COLUMBUS [65] optimizes feature selection specifically for regression models. ActiveClean [66] integrates data cleaning with learning convex models, using gradient-biased samples to identify dirty data. Brainwash [67] proposes to expedite feature engineering by recommending feature transformations. Zombie [68] speeds up data preparation by learning over smaller, actively-learned informative subsets of data during feature engineering. These approaches are bespoke for the data preprocessing portion of ML workflows and do not target end-to-end optimizations, although there is no reason they could not be integrated within HELIX.

ML and non-ML Workflow Management Tools. Here we discuss ML workflow systems, production platforms for ML, industry batch processing workflow systems, and systems for scientific workflows.

ML Workflow Management. Prior tools for managing ML workflows focus primarily on making their pipelines easier to debug. For example, Gestalt [69] and Mistique [70] both tackle the problem of model diagnostics by allowing users to inspect intermediate results. The improved workflow components in these systems could be easily incorporated within HELIX.

ML Platforms-as-Services. A number of industry frameworks [27, 71, 72, 73, 74, 75], attempt to automate typical steps in deploying machine learning by providing a Platform-as-a-Service (PaaS) capturing common use cases. These systems vary in generality — frameworks like SageMaker, Azure Studio, and MLFlow are built around services provided by Amazon, Microsoft, and Databricks, respectively, and provide general solutions for production de-

ployment of ML models for companies that in-house infrastructure. On the other hand, TFX, FBLearner Flow, and Michelangelo are optimized for internal use at Google, Facebook, and Uber, respectively. For example, TFX is optimized for use with TensorFlow, and Michelangelo is optimized for Uber’s real-time requirements, allowing production models to use features extracted from streams of live data.

The underlying “workflow” these frameworks manage is not always given an explicit representation, but the common unifying thread is the automation of production deployment, monitoring, and continuous retraining steps, thereby alleviating engineers from the labor of ad-hoc solutions. HELIX is not designed to reduce manual effort of model deployment, but rather model *development*. The workflow HELIX manages sits at a lower level than those of industry PaaS systems, and therefore the techniques it leverages are quite different.

General Batch Processing Workflow Systems. A number of companies have implemented workflow management systems for batch processing [76, 17]. These systems are not concerned with runtime optimizations, and rather provide features useful for managing large-scale workflow complexity.

Scientific Workflow Systems. Some systems address the significant mental and computational overhead associated with scientific workflows. VisTrails [77] and Kepler [78] add provenance and other metadata tracking to visualization-producing workflows, allowing for reproducibility, easier visualization comparison, and faster iteration. Other systems attempt to map scientific workflows to cluster resources [79]. One such system, Pegasus [80], also identifies reuse opportunities when executing workflows. The optimization techniques employed by all systems discussed leverage reuse in a simpler manner than does HELIX, since the workflows are coarser-grained and computation-heavy, so that the cost of loading cached intermediate results can be considered negligible.

Intermediate Results Reuse. The OEP/OMP problems within HELIX are reminiscent of classical work on view materialization in database systems [81], but operates at a more coarse-grained level on black box operators. However, the reuse of intermediate results within ML workflows differs from traditional database view materialization in that it is less concerned with fine-grained updates, and instead treats operator outputs as immutable black-box units due to the complexity of the data analytics operator. COLUMBUS [65] focuses on caching feature columns for feature selection exploration within a single workflow. ReStore [82] manages reuse of intermediates across dataflow programs written in Pig [83], while Nectar [84] does so across DryadLINQ [85] workflows. Jindal et al. [86] study SQL subexpression materialization within a single workflow with many subqueries. Perez et al. [87] also study SQL subexpression materialization, but in an inter-query fashion that

uses historical data to determine utility of materialization for future reuse. In the same vein, Mistique [70] and its spiritual predecessor Sherlock [88] use historical usage as part of their cost models for adaptive materialization. HELIX shares some similarities with the systems above but also differs in significant ways. Mistique [70], Nectar [84], and ReStore [82] share the goal of efficiently reusing ML workflow intermediates with HELIX. However, the cost models and algorithms proposed in these systems for deciding what to reuse do not consider the operator/subquery dependencies in the DAG and make decisions for each operator independently based on availability, operator type, size, and compute time. We have shown in Figure 5.1 that decisions can have cascading effects on the rest of the workflow. The reuse problems studied in COLUMBUS [65] and Jindal et al. [86] differ from ours in that they are concerned with decomposing a set of queries Q into subqueries and picking the minimum cost set of subqueries to cover Q . The queries and subqueries can be viewed as a bipartite graph, and the optimization problem can be cast as a SET COVER. They do not handle iteration but rather efficient execution of parallel queries. Furthermore, the algorithms for choosing what to materialize in Mistique [70] and Perez et al. [87] use historical data as signals for likelihood of reuse in the future, whereas our algorithm directly uses projected savings for the next iteration based on the reuse plan algorithm. Their approaches are reactive, while ours is proactive.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

We presented HELIX, a declarative system aimed at accelerating iterative ML application development. In addition to its user friendly, flexible, and succinct programming interface, HELIX tackles two major optimization problems, namely OPT-EXEC-PLAN and OPT-MAT-PLAN, that together enable cross-iteration optimizations resulting in significant run time reduction for future iterations. We devised a PTIME algorithm to solve OPT-EXEC-PLAN by using a reduction to MAX-FLOW. We showed that OPT-MAT-PLAN is NP-HARD and proposed a light-weight, effective heuristic for this purpose. We evaluated HELIX against DeepDive and KeystoneML on workflows from social sciences, NLP, computer vision, and natural sciences that vary greatly in characteristics to test the versatility of our system. We found that HELIX supports a variety of diverse machine learning applications with ease and provides *40-60% cumulative run time reduction* on complex learning tasks and *nearly an order of magnitude reduction* on simpler ML tasks compared to both DeepDive and KeystoneML. While HELIX is implemented in a specific way, the techniques and abstractions presented in this work are general-purpose; other systems can enjoy the benefits of HELIX’s optimization modules through simple wrappers and connectors.

In future work, we aim to further accelerate iterative workflow development via introspection and querying across workflow versions over time, automating trimming of redundant workflow nodes, as well as auto-suggestion of workflow components to aid workflow development by novices. Specifically, HELIX is capable of tracing specific features in the ML model to the operators in the DAG. This allows information about feature importance learned in the ML model to be used directly to prune the DAG. In addition, the materialization and reuse techniques we proposed can be extended to optimize parallel executions of similar workflows.

REFERENCES

- [1] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald, “Declarative machine learning—a classification of basic properties and types,” *arXiv preprint arXiv:1605.05826*, 2016.
- [2] M. A. Munson, “A study on the importance of and time spent on different modeling steps,” *ACM SIGKDD Explorations Newsletter*, vol. 13, no. 2, pp. 65–71, 2012.
- [3] X. Ren, J. Shen, M. Qu, X. Wang, Z. Wu, Q. Zhu, M. Jiang, F. Tao, S. Sinha, D. Liem et al., “Life-inet: A structured network-based knowledge exploration and analytics system for life sciences,” *Proceedings of ACL 2017, System Demonstrations*, pp. 55–60, 2017.
- [4] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.
- [5] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, “Line: Large-scale information network embedding,” in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.
- [6] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, “Keystoneml: Optimizing pipelines for large-scale advanced analytics,” in *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 2017, pp. 535–546.
- [7] C. Zhang, “Deepdive: A data management system for automatic knowledge base construction,” Ph.D. dissertation, Citeseer, 2015.
- [8] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education, 2006.
- [9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi et al., “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.
- [10] D. Team, “Deeplearning4j: Open-source distributed deep learning for the jvm,” *Apache Software Foundation License*, vol. 2, 2016.
- [11] W. Rasband, “Imagej: Image processing and analysis in java,” *Astrophysics Source Code Library*, 2012.
- [12] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran, “Helix: Accelerating human-in-the-loop machine learning (demo paper),” *Proceedings of the VLDB Endowment*, 2018.

- [13] D. Xin, S. Macke, L. Ma, R. Ma, S. Song, and A. Parameswaran, “Helix: Holistic optimization for accelerating iterative machine learning,” *Technical Report <http://data-people.cs.illinois.edu/helix-tr.pdf>*, 2018.
- [14] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran, “Accelerating human-in-the-loop machine learning: Challenges and opportunities (vision paper),” in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, ser. DEEM’18. ACM, 2018.
- [15] A. Karpathy and L. Fei-Fei, “Deep visual-semantic alignments for generating image descriptions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3128–3137.
- [16] J. Lin and D. Ryaboy, “Scaling big data mining infrastructure: the twitter experience,” *ACM SIGKDD Explorations Newsletter*, vol. 14, no. 2, pp. 6–19, 2013.
- [17] R. Sumbaly, J. Kreps, and S. Shah, “The big data ecosystem at linkedin,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1125–1134.
- [18] D. Xin, L. Ma, S. Song, and A. Parameswaran, “How developers iterate on machine learning workflows—a survey of the applied machine learning literature,” *KDD IDEA Workshop*, 2018.
- [19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.
- [20] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, “The stanford corenlp natural language processing toolkit.” in *ACL (System Demonstrations)*, 2014, pp. 55–60.
- [21] D. Team et al., “Deeplearning4j: Open-source distributed deep learning for the jvm,” *Apache Software Foundation License*, vol. 2.
- [22] R. Kohavi, “Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 202–207.
- [23] E. Meijer, B. Beckman, and G. Bierman, “Linq: Reconciling object, relations and xml in the .net framework,” in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’06. New York, NY, USA: ACM, 2006, pp. 706–706.
- [24] J. Rosen, “Pyspark internals.”

- [25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., “Scikit-learn: Machine learning in python,” *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.
- [26] X. Meng, J. Bradley, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen et al., “Mllib: Machine learning in apache spark,” 2016.
- [27] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc et al., “Tfx: A tensorflow-based production-scale machine learning platform,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1387–1395.
- [28] “Scikit-learn user guide,” http://scikit-learn.org/stable/user_guide.html.
- [29] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, “API design for machine learning software: experiences from the scikit-learn project,” in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.
- [30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.
- [31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al., “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv preprint arXiv:1603.04467*, 2016.
- [32] A. Paszke, S. Gross, S. Chintala, and G. Chanan, “Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration,” 2017.
- [33] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in neural information processing systems*, 2011, pp. 693–701.
- [34] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, “Parallelized stochastic gradient descent,” in *Advances in neural information processing systems*, 2010, pp. 2595–2603.
- [35] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.
- [36] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM computing surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.
- [37] A. M. Pitts, “Operationally-based theories of program equivalence,” *Semantics and Logics of Computation*, vol. 14, p. 241, 1997.
- [38] A. D. Gordon, “A tutorial on co-induction and functional programming,” in *Functional Programming, Glasgow 1994*. Springer, 1995, pp. 78–95.

- [39] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- [40] M. Yannakakis, “On a class of totally unimodular matrices,” *Mathematics of Operations Research*, vol. 10, no. 2, pp. 280–304, 1985.
- [41] D. S. Hochbaum and A. Chen, “Performance analysis and best implementations of old and new algorithms for the open-pit mining problem,” *Operations Research*, vol. 48, no. 6, pp. 894–914, 2000.
- [42] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.
- [43] “Deepdive census example,” <https://github.com/HazyResearch/deepdive/tree/master/examples/census>.
- [44] C. De Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang, “Deepdive: Declarative knowledge base construction,” *SIGMOD Rec.*, vol. 45, no. 1, pp. 60–67, June 2016. [Online]. Available: <http://doi.acm.org/10.1145/2949741.2949756>
- [45] D. Dheeru and E. Karra Taniskidou, “{UCI} machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [47] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, “Systemml: Declarative machine learning on mapreduce,” in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 231–242.
- [48] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, “Optiml: an implicitly parallel domain-specific language for machine learning,” in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 609–616.
- [49] M. Weimer, T. Condie, R. Ramakrishnan et al., “Machine learning in scalops, a higher order cloud computing language,” in *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, vol. 9, 2011, pp. 389–396.
- [50] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, “The architecture of scidb,” in *International Conference on Scientific and Statistical Database Management*. Springer, 2011, pp. 1–16.
- [51] S. Owen, R. Anil, T. Dunning, and E. Friedman, “Mahout in action,” 2012.
- [52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: a framework for machine learning and data mining in the cloud,” *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

- [53] J. Langford, L. Li, and A. Strehl, “Vowpal wabbit online learning project,” 2007.
- [54] A. Damien et al., “Tflearn,” 2016.
- [55] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li et al., “The madlib analytics library: or mad skills, the sql,” *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1700–1711, 2012.
- [56] X. Feng, A. Kumar, B. Recht, and C. Ré, “Towards a unified architecture for in-rdbms analytics,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 325–336.
- [57] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick, “Hybrid in-database inference for declarative information extraction,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 517–528.
- [58] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab, “Weld: A common runtime for high performance data analytics,” in *Conference on Innovative Data Systems Research (CIDR)*, 2017.
- [59] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, “An architecture for compiling udf-centric workflows,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1466–1477, 2015.
- [60] A. Kumar, J. Naughton, and J. M. Patel, “Learning generalized linear models over normalized data,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1969–1984.
- [61] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, “Towards linear algebra over normalized data,” *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1214–1225, 2017.
- [62] D. Olteanu and M. Schleich, “F: regression models over factorized views,” *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1573–1576, 2016.
- [63] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu, “To join or not to join?: Thinking twice about joins before feature selection,” in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 19–34.
- [64] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, “Snorkel: Rapid training data creation with weak supervision,” *arXiv preprint arXiv:1711.10160*, 2017.
- [65] C. Zhang, A. Kumar, and C. Ré, “Materialization optimizations for feature selection workloads,” *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 1, p. 2, 2016.
- [66] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, “Activeclean: Interactive data cleaning while learning convex loss models,” *arXiv preprint arXiv:1601.03797*, 2016.

- [67] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang, “Brainwash: A data system for feature engineering.” in *CIDR*, 2013.
- [68] M. R. Anderson and M. Cafarella, “Input selection for fast feature engineering,” in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016, pp. 577–588.
- [69] K. Patel, N. Bancroft, S. M. Drucker, J. Fogarty, A. J. Ko, and J. Landay, “Gestalt: integrated support for implementation and analysis in machine learning,” in *Proceedings of the 23rd annual ACM symposium on User interface software and technology*. ACM, 2010, pp. 37–46.
- [70] M. Vartak, J. M. da Trindade, S. Madden, and M. Zaharia, “Mistique: A system to store and query model intermediates for model diagnosis,” in *Proceedings of the 2018 ACM International Conference on Management of Data*, 2018.
- [71] J. Dunn, “Introducing fblearner flow: Facebook’s ai backbone,” 2018 (accessed October 8, 2018), <https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/>.
- [72] J. Barnes, “Azure machine learning microsoft azure essentials,” 2015.
- [73] “Meet michelangelo: Uber’s machine learning platform,” 2017 (accessed October 8, 2018), <https://eng.uber.com/michelangelo/>.
- [74] *Amazon SageMaker Developer Guide*, 2018 (accessed October 8, 2018), <https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf>.
- [75] M. Zaharia, “Introducing mlflow: an open source machine learning platform,” 2018 (accessed October 8, 2018), <https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html>.
- [76] M. Beauchemin, “Airflow: a workflow management platform,” 2015 (accessed October 8, 2018), <https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8>.
- [77] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, “Vistrails: visualization meets data management,” in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 745–747.
- [78] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the kepler system,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.
- [79] J. Yu and R. Buyya, “A taxonomy of workflow management systems for grid computing,” *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 171–200, 2005.

- [80] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, “Pegasus: Mapping scientific workflows onto the grid,” in *Grid Computing*. Springer, 2004, pp. 11–20.
- [81] R. Chirkova, J. Yang et al., “Materialized views,” *Foundations and Trends® in Databases*, vol. 4, no. 4, pp. 295–405, 2012.
- [82] I. Elghandour and A. Abounaga, “Restore: reusing results of mapreduce jobs,” *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 586–597, 2012.
- [83] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: a not-so-foreign language for data processing,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1099–1110.
- [84] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, “Nectar: Automatic management of data and computation in datacenters.” in *OSDI*, vol. 10, 2010, pp. 1–8.
- [85] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, “Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.” in *OSDI*, vol. 8, 2008, pp. 1–14.
- [86] A. Jindal, K. Karanasos, S. Rao, and H. Patel, “Selecting subexpressions to materialize at datacenter scale,” *Proceedings of the VLDB Endowment*, vol. 11, no. 7, pp. 800–812, 2018.
- [87] L. L. Perez and C. M. Jermaine, “History-aware query optimization with materialized intermediate views,” in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 520–531.
- [88] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia, “Supporting fast iteration in model building,” in *NIPS Workshop LearningSys*, 2015.