

© 2021 Stephen Thomas Macke

LEVERAGING DISTRIBUTIONAL CONTEXT FOR
SAFE AND INTERACTIVE DATA SCIENCE AT SCALE

BY

STEPHEN THOMAS MACKE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Assistant Professor Aditya Parameswaran, Chair
Associate Professor Hari Sundaram
Associate Professor Hanghang Tong
Dr. Alex Beutel, Google

ABSTRACT

Data science is an iterative, exploratory, and ad-hoc process performed by individuals and teams possessing increasingly varied backgrounds and skill-sets. As such, we need data science to be *interactive*, so that data scientists are not bottlenecked when trying out new hypotheses or confirming existing ones. Moreover, data science must be *safe*, ensuring that data scientists, especially those with limited programming or analysis experience, avoid making incorrect inferences. Safety and interactivity are typically at odds with one another, since various notions of safety often eschew “shortcuts” that make working with large-scale data tractable. In this dissertation, we aim to meet the dual objectives of interactivity and safety *at scale* by leveraging *distributional context*—specifically the distributions of the data and the operations performed by data scientists.

We apply this “recipe” to five different key data science settings: (i) for *machine learning development*, we provide context-aware caching algorithms that allow model developers to benefit from interactive iteration times during model development, while not requiring error-prone manual tracking of reusable intermediates; (ii) for *visualization search*, we develop context-aware sampling algorithms that support interactive search for patterns in visualizations, while ensuring that the results meet rigorous quality guarantees; (iii) for *browsing*, we develop workload-aware learned Bloom filters optimized for multidimensional data that allow analysts to quickly identify records that have been examined before, all while guarding against false negatives; (iv) for *report generation*, we develop context-aware aggregate approximation algorithms that provide rigorous distribution-aware confidence intervals around aggregates, while ensuring that the intervals are “tighter”, allowing analysts to make decisions sooner; and (v) finally, for error-prone interactions in *computational notebooks*, we demonstrate approximate lineage-capture techniques that warn data scientists of unsafe cell executions for many cases encountered in practice.

To my parents, Edward and Sally Macke, for their unconditional love and encouragement.

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Professor Aditya Parameswaran, for extensive guidance and mentorship throughout my graduate studies. Aditya set the standard for how to behave like a scientist, and did so with the appropriate mixture of humility and aplomb. Any fruit borne during my Ph.D. is due to Aditya's willingness to nurture my ideas.

I furthermore owe particular thanks to the members of my Ph.D. committee; Prof. Hari Sundaram, Prof. Hanghang Tong, and Dr. Alex Beutel, for reading my dissertation and providing valuable feedback.

I am extraordinarily lucky to have received guidance from a great many accomplished and talented academics. Alex Beutel in particular taught me a great deal about how to apply the scientific method efficiently, but I am generally fortunate to have collaborated and interacted with Jiawei Han, Ronitt Rubinfeld, Ed Chi, Joe Hellerstein, Joey Gonzalez, Tim Kraska, Ilias Diakonakolis, Derek Cheng, Venky Ganti, Alkis Polyzotis, Maheswaran Sathiamoorthy, Sarah Chasins, and many others.

I recognize that not everyone is so lucky as me to have behind their graduate school journey a network of unwavering support; for me, this came in large part from my parents, my siblings, my partner, and my cat, without whom none of this would be possible.

Special thanks to the passionate and tenacious students with whom I was fortunate to cross paths over the course of my graduate studies; for serving as inspiration, moral support, and for sharing the wonderful and incomparable experience of graduate school: Ahmed El-Kishky, Akash Das Sarma, Andrew Head, Andrew Kuznetsov, Angela Lee, Aston Zhang, Brandon Norick, Daniel Rothchild, Devin Petersohn, Doris Lee, Doris Xin, Edward Xue, Eyal Sela, Fangbo Tao, Huan Gui, Honglei Zhuang, Jaewoo Kim, Jerry Song, Jialin Liu, Jingjing Wang, Joon Sung Park, Kelly Mack, Litian Ma, Liqi Xu, Mangesh Bendre, Maryam Aliakbarpour, Michael Whittaker, Paras Jain, Ray Gong, Rolando Garcia, Sajjadur Rahman, Shi Zhi, Silu Huang, Tana Wattanawaroon, Tarique Siddiqui, Ti-Chung Cheng, Vipul Venkataraman, William Ma, Xinyan Zhou, Yifan Wu, Yihan Gao, Yiming Zhang, Yu Shi, and many others.

Finally, to my loving partner, Sophie: thank you for walking with me hand-in-hand during this journey and for leading by example with your persistence.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Background	1
1.2	Challenges Balancing Safety and Interactivity	1
1.3	Our Vision	3
1.4	Contributions and Outline	3
CHAPTER 2	LITERATURE REVIEW	6
2.1	Machine Learning Workflows	6
2.2	Browsing and Bitmap Indexes	8
2.3	Interactive Analytics	9
2.4	Computational Notebooks	11
CHAPTER 3	SAFE REUSE IN MACHINE LEARNING WORKFLOWS	13
3.1	Motivation	13
3.2	System Architecture	14
3.3	Programming Interface	18
3.4	Compilation and Representation	25
3.5	Optimization	27
3.6	Empirical Evaluation	35
3.7	Summary	47
CHAPTER 4	SAFE BROWSING WITH LEARNED BLOOM FILTERS	49
4.1	Overview	49
4.2	Learning Filters	50
4.3	Challenges and Optimizations	51
4.4	Experimental Results	53
4.5	Summary	54
CHAPTER 5	SAFE APPROXIMATION FOR VISUAL EXPLORATION	55
5.1	Motivation	55
5.2	Problem Formulation	56
5.3	The HistSim Algorithm	61
5.4	The FastMatch System	74
5.5	Empirical Study	81
5.6	Summary	88

CHAPTER 6	SAFE APPROXIMATION FOR REPORT GENERATION	89
6.1	Motivation	89
6.2	DBMS Error Bound Integration	90
6.3	Fixing Bounder Pathologies	101
6.4	System Considerations	104
6.5	Empirical Study	111
6.6	Summary	123
CHAPTER 7	APPROXIMATE LINEAGE FOR SAFE NOTEBOOK INTERACTIONS . .	125
7.1	Motivation	125
7.2	Architecture Overview	127
7.3	Lineage Tracking	129
7.4	Liveness and Inverse Liveness	134
7.5	Cell Highlights	139
7.6	Empirical Study	143
7.7	Summary	156
CHAPTER 8	CONCLUSION AND FUTURE WORK	157
8.1	Key Takeaways and Design Principles	157
8.2	Future Work	158
APPENDIX A	PROOFS OF SELECTED THEOREMS	160
A.1	Proof of Theorem 3.2	160
A.2	Proof of Theorem 3.3	162
A.3	Proof of Theorem 4.1	163
A.4	Proof of Theorem 5.1	165
A.5	Proof of Theorem 6.1	166
A.6	Proof of Theorem 6.2	167
APPENDIX B	HML SPECIFICATIONS	171
B.1	Summary of HML Semantics	171
B.2	HML Grammar	172
APPENDIX C	SYSTEM EXTENSIONS	173
C.1	Generalizing FASTMATCH to Additional Queries	173
C.2	Different Types of FASTMATCH Guarantees	175
C.3	Handling Arbitrary Expressions in FastFrame	175
REFERENCES	178

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

Data in recent times has grown to scales that tax the computational capabilities of many companies and individuals, thereby impeding analysis and hindering the ability to extract valuable insights. Data scientists, whose skills are already stretched thin from having to wear many hats (developer, modeler, communicator), bear the brunt of the pain incurred from working with massive data.

While no kind of software development occurs in a vacuum, data science in particular is highly *interactive*, with the analysis and development process adaptively guided while browsing, plotting visualizations, generating reports, and modeling data. The types of unpredictable workloads shouldered by data scientists are particularly sensitive to problems stemming from scale: while certain computing tasks can be run “in batch” while out to lunch, data science tasks are fluid and exploratory in nature, requiring a human-in-the-loop for continually monitoring results and incrementally testing, rejecting, and refining hypotheses. In fact, recent work suggests that latencies greater than 500ms cause significant frustration for end-users and lead them to test fewer hypotheses and potentially identify fewer insights [1].

The demanding nature of data science at scale typically requires “cutting through the data jungle” to accomplish even though most basic of tasks, necessitating shortcuts to help cope with the scale of the data. To alleviate pain points of common data science tasks, recent work has developed strategies based on sampling [2, 3, 4, 5, 6] and deterministic approximation [7] to aid with visualization and report generation, caching and smart reuse [8, 9, 10] to optimize machine learning workflows, smart indexing strategies [11] to aid with browsing of records that match desired criteria, and strategies for provenance capture [12, 13, 14] to make computational notebooks less error prone. However, facilitating these tasks at scale while optimizing for interactivity, and simultaneously ensuring safety via correctness guarantees has remained an ideal largely beyond reach.

1.2 CHALLENGES BALANCING SAFETY AND INTERACTIVITY

A common observation is that existing systems tend to optimize for either safety or interactivity at the expense of the other. We now illustrate such issues in the context of machine learning development, data exploration and visualization, and computational notebooks.

Machine Learning Development. During machine learning model development, data scientists iterate on existing ML workflows via small refinements, thereby admitting reuse opportunities. Previous systems have leveraged such reuse opportunities [8, 9, 10] during machine learning

training, but for intermediates created during batch executions (e.g., for training that runs on a schedule), as opposed to the interactive development performed by data scientists. While nothing in principle prevents data scientists from performing their own manual caching and reuse, this kind of manual workflow management is highly error-prone and cumbersome. As a result, modelers tend to recompute their entire workflows from scratch, which slows development and hinders interactivity.

Index Structures for Browsing. *Browsing* is a common task whereby data scientists examine records that match desired criteria. Bitmap index structures are commonly used to reduce the amount of data fetched from storage media during browsing; unfortunately, for large datasets, the index is itself oftentimes too large to fit in memory (especially if over multiple attributes) and must be paged in from storage media, thereby increasing I/O cost. Therefore, novel approaches for compressing bitmap index structures are paramount for enabling browsing at scale, and such techniques should be *lossless* in order to ensure safety; i.e., to ensure that records are not mistakenly omitted from the result set.

Visual Exploration. Data scientists often attempt to generate visualizations that *match desired criteria* [15]. Histograms are a very common visualization type for which users attempt to slice their data in order to generate matches, but the process of iterative “generate and test” occupies a large part of visual data analysis [16, 17, 18], and is often cumbersome and time consuming, especially on very large datasets that are increasingly the norm.

Sampling is a natural approach to cope with data at scale, but introduces error, compromising safety in the form of the fidelity of the results. To rectify this and bolster safety, general sampling-based algorithms with strong guarantees for distinguishing of top- k “desirable arms” [19] (matching histograms in our setting), have been developed. Unfortunately, such techniques require too many samples to be practical.

Report Generation. AVG and SUM aggregations are fundamental operations powering OLAP and report generation; ensuring they run quickly is therefore of great importance. Over the years, many attempts have been made to accelerate these operations. One category of optimizations leverages approximation, using sampling-based techniques in order to estimate the aggregate on a subset of the full data [2, 3, 4, 5, 6, 20, 21, 22, 23, 24].

Prior work relies on two kinds of classes of approximation techniques: *asymptotic* techniques that use properties of the sample to compute error bounds that give accuracy guarantees as the size of the sample approaches infinity [2, 3, 4, 21, 23], and sample-agnostic techniques that leverage intrinsic properties of the data known a priori in order to compute error bounds that depend only on the size of the sample taken, as opposed to properties of the sample itself [2, 6, 20, 24]. In both cases, the goal of an error bound quantification procedure is to ❶ *minimize the size of the error bounds* (so that the query can terminate with fewer samples and therefore more quickly), but ❷ *subject to the*

constraint that they enclose the true result following an exact computation of the aggregate. ❶ helps to minimize latency, while ❷ ensures that query results fall within acceptable error tolerances.

These twin goals are at odds with one another. The aforementioned asymptotic techniques prioritize ❶, but do so at the cost of safety. Conservative approximate query processing (AQP), on the other hand, facilitates strong guarantees at the expense of ❷, as the traditional techniques based on, e.g., Hoeffding’s inequality give error bounds that are far too loose to be applicable in a data science setting where interactive latencies are required.

Computational Notebooks. The programming environment favored by modern data scientists for undertaking the above tasks is the *computational notebook*. Computational notebooks such as Jupyter [25] keep intermediate programming state in memory, thereby breaking out of the batch programming paradigm and enabling rapid prototyping and exploration. Despite these conveniences, notebooks’ hidden programming state accumulates over time and can be difficult to reason about, leading to counter-intuitive and error-prone interactions, hindering reproducibility and confusing users [26]. Thus, the very environment preferred for undertaking data science tasks itself compromises on safety in order to facilitate interactivity.

1.3 OUR VISION

We envision systems capable of *leveraging distributional context*, i.e., of the data or workload in question, that bolster the reliability of common data science tasks while preserving interactive latencies, or vice versa (reducing latencies without compromising safety).

To illustrate the basic idea, consider an example from sampling-based AQP. Suppose we have a query q_1 that asks whether the average of column C is higher or lower than 0, versus query q_2 that directly asks for the average of column C . Answering either query exactly would have the same cost, while q_1 can be less costly to approximate than q_2 in a way that gives correctness guarantees. This is an example of *leveraging workload context* to enable *safe* (i.e., due to guarantees) interactivity at scale, since we are able to use properties of q_1 to terminate sooner than if we were to use exact processing (which would ascribe identical cost to q_1 and q_2). To give an example of leveraging *data context*, note that q_2 might be easier to compute (i.e., require fewer samples) if C has low variance, compared to if C has high variance.

1.4 CONTRIBUTIONS AND OUTLINE

In this dissertation, we study techniques for moving toward both safety and interactivity in five different data science scenarios. The remaining chapters are organized as follows:

Chapter	Safety Aspect(s)	Interactivity Aspect(s)	Context
Chapter 3	Avoid use of invalidated workflow intermediates	Optimal reuse of ML workflow intermediates	Similarity between adjacent workflow iterations
Chapter 4	Enforce no false negatives for index lookups	Compressed index sizes can fit in memory more easily	Differences between indexed data and queries for out-of-index data
Chapter 5	Guarantees on separation and reconstruction	Reduced latencies thanks to early stopping	Ability to prune poor matches more quickly than good matches
Chapter 6	Strong guarantees on error bounds	Reduced latencies thanks to early stopping	Ability to safely compute tighter bounds for low-variance data
Chapter 7	Prevent incorrect stale cell executions	Bound lineage overhead using trace-once semantics	Observe that trace-once semantics suffice for most common notebook usage patterns

Table 1.1: Breakdown of dissertation themes by chapter.

In **Chapter 2, Literature Review**, we survey work related to various data science tasks considered in this dissertation along the axes of safety and interactivity, calling to particular attention prior examples that leverage data or workload characteristics to benefit one or the other.

Following the literature review, we have the core content chapters of this dissertation:

In **Chapter 3, Safe Reuse In Machine Learning Workflows**, we develop HELIX, a system capable of jointly managing workflow state and optimizing across workflow iterations. Pushing the workflow state into the HELIX system enables safety, and smart policies for materialization and reuse benefit from the incremental workflow aspect in order to enable interactivity at scale. The contents of **Chapter 3** have appeared in our VLDB’19 paper [27]. This was work led by Doris Xin, but our contribution was the algorithm for optimal reuse.

In **Chapter 4, Safe Browsing with Learned Bloom Filters**, we demonstrate workload-aware or *learned* Bloom filters that use less space than their traditional counterparts to enable safe browsing (without false negatives for lookups) over massive sets of multidimensional records. The contents of this chapter have appeared in our ML for Systems workshop paper [28] at NeurIPS’18.

In **Chapter 5, Safe Approximation for Visual Exploration**, we develop a system, FASTMATCH, capable of sampling in order to rapidly find the k histograms that best match an analyst-provided target. FASTMATCH uses an *adaptive* technique to prune confidently matching / non-matching

histograms from the sampling routine, coupled with a *lookahead* technique that exploits locality when querying bitmap indexes to find blocks containing samples for still-uncertain histograms. By facilitating a correctness guarantee, the system guarantees safety; thanks to the extensive optimizations that enable matching queries to be performed at fast latencies, the system facilitates interactivity at scale. The contents of **Chapter 5** have appeared in our VLDB’18 paper [29].

In Chapter 6, Safe Approximation for Report Generation, we jointly improve safety and interactivity of report generation at large scales by providing a set of data-aware sampling techniques for computing aggregates. These techniques are designed to rectify common issues faced by traditional conservative AQP, and are able to leverage properties of data learned during sampling to compute error bounds, but *without* sacrificing strong probabilistic guarantees. We develop these distribution-sensitive report generation techniques within the context of a system, FASTFRAME, which extends the FASTMATCH architecture from **Chapter 5**. The contents of this chapter have appeared in our ICDE’21 paper [30].

In Chapter 7, Approximate Lineage for Safe Notebook Interactions, we develop NBSAFETY, a custom kernel built on top of the Jupyter runtime which combines runtime tracing and static analysis to track at fine granularity the lineage of variables and other state introduced by the data scientist. We show that NBSAFETY is able to effectively use this lineage to provide hints and warnings about cells that may be unsafe to execute. Tracking this lineage can introduce significant overhead, however. Fortunately, we show that we are able to retain virtually all of NBSAFETY’s safety benefits by tailoring the tracked lineage to characteristics specific to computational notebooks. To do so, we introduce *trace-once* semantics in order to guarantee that the tracked lineage accumulates only in proportion to the amount of code written by the data scientist, and never grows without bound. The contents of **Chapter 7** will have appeared in our VLDB’21 paper [31].

Chapters 3 to 7 constitute the core chapters of this dissertation. **Table 1.1** summarizes the themes of safety, interactivity, and distributional awareness within the context of each of these chapters.

In Chapter 8, Conclusion and Future Work, we conclude and suggest avenues for further exploration based on the the core ideas outlined in this dissertation.

CHAPTER 2: LITERATURE REVIEW

In this chapter, we survey related directions toward enabling safe and interactive data science. Although the majority of prior efforts have focused on reducing latency of data science tasks (and thereby improving interactivity), we call to particular attention efforts related to safety aspects wherever relevant, as well as how workload or data characteristics are used to improve aspects of interactivity under safety constraints, or vice versa. We will typically categorize safety concerns along two axes: those that ensure that data science-related queries are correctly formulated, which we deem *query safety*; and those that, given a correctly formulated query, ensure that the underlying system returns a correct result (within acceptable error tolerance), which we deem *result safety*.

2.1 MACHINE LEARNING WORKFLOWS

We now survey prior work on machine learning systems. We segment our review along two axes: *user-facing* systems that aim to improve the development process, and *production-focused* systems that attempt to automate aspects of machine learning productionization. We shall see that user-facing systems typically focus on improving interactivity with relatively little consideration given to safety, while production systems typically focus on safety, with little attention given to developer convenience. Furthermore, we are aware of no prior work along either axis that considers the interplay between safety and interactivity in the context of the highly iterative and incremental process of machine learning development.

User-Facing ML Systems. Prior efforts toward user-facing systems for ML workflow development have focused on reducing latency while improving expressiveness of such systems. These twin goals can also be framed in terms of safety and interactivity: latency reductions tighten the feedback cycle and improve interactivity, while improvements to expressiveness (typically facilitated by providing a predefined set of declarative ML building blocks) reduce time needed for development and furthermore reduce errors during the development process by allowing for reuse of existing, well-tested software components. However, safety is typically not the primary goal of such systems, but a byproduct of the convenience they offer.

Expressiveness has traditionally been facilitated by making it easier to specify ML workflows *declaratively*. Boehm et al. categorize declarative ML systems into three groups based on the usage: *declarative ML algorithms*, *ML libraries*, and *declarative ML tasks* [32]. Systems that support *declarative ML algorithms*, such as TensorFlow [33], SystemML [34], OptiML [35], ScalOps [36], and SciDB [37], allow ML experts to program new ML algorithms, by declaratively specifying linear algebra and statistical operations at higher levels of abstraction.

ML libraries, such as Mahout [38], Weka [39], GraphLab [40], Vowpal Wabbit [41], MLlib [42] and Scikit-learn [43], provide simple interfaces to optimized implementations of popular ML algorithms. TensorFlow has also recently started providing TFLearn [44], 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. DeepDive [9, 10] and KeystoneML [8] also fall into this group of systems, which perform workflow-level optimizations to reduce end-to-end execution time, thereby improving interactivity. Declarative ML task systems can seamlessly make use of improvements in ML library implementations, such as MLlib [42], CoreNLP [45] and DeepLearning4j [46], within UDF calls. Unlike declarative ML algorithm systems, that are targeted at ML experts and researchers, these systems focus on end-users of ML. They improve interactivity for their specific respective tasks at the cost to flexibility.

In general, all of the above systems help improve interactivity via reduced latencies, but they tend to ignore the highly error-prone and interactive aspects of iterative ML workflow *development*. The focus is more on end-to-end optimization of a single workflow run expressed via convenient high-level primitives, as opposed to optimizations and error reductions across multiple runs.

Production-Focused ML Systems. A number of industry frameworks [47, 48, 49, 50, 51, 52, 53], 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 [51], Azure Studio [49], and MLFlow [52] are built around services provided by Amazon, Microsoft, and Databricks, respectively, and along with Apache Airflow [53] provide general solutions for production deployment of ML models for companies that in-house infrastructure. On the other hand, TFX [47], FBLearner Flow [48], and Michelangelo [50] 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 aspects of machine learning production pipelines: allocation and fault tolerance of cluster resources, production deployment, monitoring dashboards, and continuous retraining steps, to name a few. The purpose of this automation is to free up engineering resources from the toil and manual labor of tailoring bespoke pipelines for each new business use case, and to provide a set of “guard rails” for catching bugs and other production issues via workflow schema validation, monitoring, and gradual rollout.

While such systems pay lip service to safety (enabling aspects of, for example, our notion of query safety by requiring workflow components to undergo schema validation), they still tend to fall under the umbrella of batch computing rather than interactive computing, as their focus is

productionization rather than development. Furthermore, one could consider optimizations enabled by the aforementioned systems as helping to facilitate interactivity, but such optimizations tend to focus on management, allocation, and efficient use of cluster resources, rather than on making the life of the data scientist easier. For example, while FBLearner Flow [48] will attempt to memoize workflow intermediates for operators whose inputs have not changed between runs, it is still the responsibility of the user to ensure that state not captured by the system (external configuration, data living in hive tables) has not changed to avoid an incorrect use of a previously memoized result. In this way, any benefits from improved interactivity are offset by reductions to safety, from the perspective of individual users.

With this in mind, our contribution in [Chapter 3](#) is to show how interactivity and safety of machine learning systems can be jointly optimized by leveraging workflow characteristics (in the form of similarities inherent in adjacent workflow iterations). By freeing users of the burden of choosing what workflow intermediates to cache and remembering when such intermediates can be safely reused, we show how to create a highly optimized ML development feedback loop.

We survey additional tools for workflow management when we cover computational notebooks in [Section 2.4](#), which share many similarities in terms of safety and interactivity challenges associated with machine learning workflows.

2.2 BROWSING AND BITMAP INDEXES

To facilitate browsing records in tabular data that satisfy some set of desired criteria, a number of different index structures have traditionally been employed. From a safety standpoint, any index structure should ensure that records matching filter criteria are never excluded from the result set; virtually all index structures make this guarantee. From an interactivity standpoint, the most important aspect of an index structure is its size, as queries to such index structures are typically much faster when the indexes fit in memory. Bitmap indexes are commonly employed for interactive analytics in particular [54, 55], necessitating the development of a number of compression techniques, as the straightforward, naïve bitmap encoding is too cumbersome to be useful in practice [56]. We survey these approaches along two axes: techniques that leverage *data characteristics*, and techniques that leverage *workload characteristics*.

Depending on data characteristics, different bitmap indexing schemes lead to different compression tradeoffs. For example, integers that are dense within some range might be represented by a straightforward bitmap structure (possibly paired with a starting point, if other than 0); integers that are sparse in a range might be represented by a packed array that stores the values proper, and a complete ranges with all contiguous integers might be represented by a pair of (start position,

end position). In fact, this was one of the insights of the Roaring bitmap format [57, 58], which adopts a hybrid encoding scheme that tries to adapt to data exhibiting multiple different sets of characteristics.

On the other hand, the query workload must also be taken into account when choosing bitmaps for optimal performance. For example, differential encoding [59] or run-length encoding (RLE) schemes [60, 61, 62] will typically perform well when querying sorted integers for set membership, while the Roaring format was shown to outperform these techniques for set membership queries involving unsorted, random integers [57, 58]. This disparity can be explained by noting that RLE schemes “store” the set members in sorted order as well, and must be decoded starting from the beginning when an item is queried — for queries to sorted integers, the same round of decoding can be shared for queries to multiple integers, while costly backtracking is necessary for queries to unsorted integers.

Bloom filters are a specialized kind of bitmap index that relax the correctness guarantees of other bitmap index structures slightly: they guarantee that all the records satisfying the desired filtering criterion will be returned, as well as some potential additional *false positive* records. As false positives can always be filtered out in a subsequent post-processing step, Bloom filters can likewise be used to facilitate result safety. Although traditional Bloom filters do not leverage workload characteristics, recent work [63, 64] has proposed *learned Bloom filters* that model the difference between the in-set data and queries for the out-of-set data, which can result in index sizes that are much smaller in some cases. Our contribution in Chapter 4 is to understand more deeply the interplay between the learned and traditional components of learned Bloom filters, and to provide a recommended set of configurations for learning Bloom filters over multidimensional data.

2.3 INTERACTIVE ANALYTICS

Interactive analytics tasks, such as visualization, report generation, and more general query processing, as well as the optimization thereof have enjoyed extensive treatment in prior work. Approaches for reducing latencies associated with such tasks are numerous; we restrict our survey to sampling-based techniques as they are the most related with Chapters 5 and 6 of this dissertation. Sampling-based interactive analytics relies on an extremely simple idea: the smaller the size of the data that are processed, the faster the processing can terminate. Such early stopping, however, can introduce error and sacrifice our notion of result safety if care is not taken. We therefore survey such techniques along two axes: those that facilitate strong guarantees regarding the correctness of the results independent of sample size, and those that only give such guarantees asymptotically, in the limit of infinite data.

Asymptotic Techniques. Central to sampling-based interactive analytics is the *confidence interval* (CI), used for computing error bounds for some quantity of interest in the result set. *Asymptotic* error bounding techniques such as bootstrap CIs [23, 65, 66] or central limit theorem (CLT)-based CIs [67, 68] make assumptions about the distribution taken by the data given a “large enough” sample size. These procedures typically give CIs that are much tighter (and therefore more useful for drawing inferences about the query results), and have enjoyed numerous applications in database and visual analytics systems [4, 69, 70, 71, 72, 73], including Aqua [74], BlinkDB [3, 75], DBO [76], and online aggregation [2], and have furthermore seen a number of DBMS-specific extensions [23, 77].

In particular, Aqua, BlinkDB, and other systems [78, 79] leverage workload characteristics to materialize offline samples or other index structures relevant to queries ahead-of-time. Although these workload-aware samples help speed up processing for downstream analytics tasks, they can be less useful for ad hoc exploration, where the workload can be unpredictable.

Furthermore, all of the aforementioned asymptotic techniques yield results that only have guaranteed error bounds w.h.p. in the limit as the size of the sample grows to infinity; i.e., they provide no real guarantees for any given finite instance, potentially leading to failures downstream. For example, consider a query that filters results post-aggregation via a HAVING clause. An AQP system could use CIs to facilitate early stopping for this query by using them to infer on which side of the HAVING threshold the various groups involved in the aggregation appear. If such a system relies on asymptotic CIs, it lacks result safety as it is prone to two serious types of error, called *subset error* and *superset error* [70], whereby certain tuples may be missing, and other tuples may appear spuriously, respectively.

Sample Size-Independent Techniques. Recognizing the downsides of asymptotic approaches, recent work [5, 6, 20, 24] has begun to adopt *conservative* error bounders, which leverage concentration inequalities to compute CIs. These procedures return bounds that follow *probably approximately correct* (PAC) [80] semantics: given $\delta \in [0, 1]$, the probability that the procedure returns lower and upper bounds $[g_\ell, g_r]$ around the approximate aggregate \hat{g} that fail to enclose the true aggregate g^* should be *at most* δ for *any* sample size (in contrast with asymptotic techniques, for which the probability converges to δ given a large enough sample). These techniques have been applied to generate visualizations that preserve visual properties [6, 24], and for incremental generation of time-series and heat-maps [5]. Similarly, Pangloss [81] computes approximate discrete distributions via the Sample+Seek approach [20].

While these techniques ensure result safety by giving a guarantee on correctness that holds independent of sample size, they can be overly conservative compared to asymptotic techniques, taking many more samples than are necessary for the guarantee. The conservativeness of these techniques can be explained by the fact that they must hold for worst-case data, and do not, in general, alter their behavior for non worst-case data [82]. Some of the aforementioned techniques for visual

analytics are able to reduce the number of samples needed to safely terminate by using some limited notions of data characteristics. For example, the techniques of Kim et al. [6] require fewer samples when computing bar chart visualizations for which the heights of the bars vary greatly. We adopt this same strategy in [Chapter 5](#) when designing sampling-based algorithms for finding visualizations that match a desired target, and we show in [Chapter 6](#) how to further leverage characteristics of the data corresponding to a single visual element (e.g., bar in a bar chart) to further reduce required samples.

2.4 COMPUTATIONAL NOTEBOOKS

We now turn our attention to computational notebooks. In many cases, computational notebooks are the preferred environment for conducting data science tasks such as exploration, visualization, and machine learning. The popularity of notebooks can be attributed to their highly interactive nature as well as the ease with which the cell-based execution model supports back references to and iteration on previously submitted code, compared to simpler REPL interfaces. However, this additional flexibility and interactivity is a double-edged sword: error-prone interactions with global notebook state are well-documented in industry and academic communities [12, 13, 26, 83, 84, 85, 86, 87, 88, 89].

In fact, notebooks face many of the same safety challenges as iterative machine learning, particularly associated with potential erroneous reuse of stale intermediates. Indeed, one does not have to squint terribly hard to view a notebook as a type of “workflow”, wherein the cells constitute workflow steps. It is thus perhaps unsurprising that prior work related to enabling safer notebook interactions has attempted to make the dependency structure in the notebook explicit via provenance capture, thereby augmenting notebooks with the benefits of existing tools for workflow management.

Provenance capture can be either *coarse-grained*, typically employed by scientific workflow systems, e.g. [90, 91, 92, 93, 94], or *fine-grained* provenance as in database systems [95, 96, 97], typically at the level of individual rows. For example, Burrito [98] tracks file and script-level coarse-grained provenance, while noWorkflow [99, 100] additionally captures finer-grained control flow dependencies, libraries, and environment variables in scripts as well as in computational notebooks [101]. These systems target post-hoc analysis of fixed parameterized scripts to understand, e.g., how changing some parameter affects the result of some experiment, but do not directly enable safer notebook interactions.

To more directly enable safer notebook interactions, the idea of treating a notebook as a workflow / dataflow computation graph with interdependent cells has been studied previously [12, 13, 14, 102]. For example, Dataflow notebooks [12] make this graph explicit by allowing users to annotate cells with their dependencies, and avoid stale usages of intermediates by forcing the re-execution of cells

whose dependencies have changed. Nodebook [13] and the Datalore kernel [14] serialize variables created by each cell and compare them with previously serialized values to determine whether such values have changed and whether subsequent cells should be re-executed, thereby enforcing a temporal ordering of cell executions. NBGATHER [84] takes a purely static approach to automatically organize notebooks using program slicing techniques, and thereby reducing non-reproducibility and errors due to messy notebooks. Vizier [102] attempts to combine cell versioning and data provenance into a cohesive notebook system with an intuitive interface, while warning users of *caveats* (i.e., possibly brittle assumptions that the analyst made about the data).

Despite the breadth of existing work related to safety aspects in computational notebooks, relatively little attention has been given to the interactivity side. In many cases, additional safety enabled by prior efforts comes at the cost of performance (e.g., in the case of solutions that serialize intermediates, such as Nodebook or Datalore), or at the cost of flexibility (e.g., in the case of Dataflow notebooks, which force users to assemble the dependency graph manually). Our contribution in [Chapter 7](#) is to provide a system that largely does not compromise on either aspect, by effectively capturing fine-grained lineage automatically for a large class of typical notebook usage patterns, used to prevent data scientists from executing cells that reference stale data.

CHAPTER 3: SAFE REUSE IN MACHINE LEARNING WORKFLOWS

In this chapter, we present, HELIX, a system for safe and interactive machine learning development. HELIX is optimized for the iterative aspect of model development, and identifies intermediate reuse opportunities by exploiting inter-iteration workflow similarities, a form of workload context. After motivating its design, we introduce its components and evaluate it on several real ML workflows.

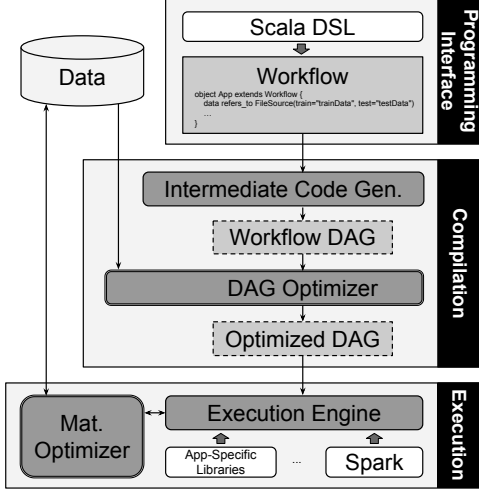
3.1 MOTIVATION

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, thereby prompting a number of efforts to make machine learning more declarative and to speed up the model training process [32].

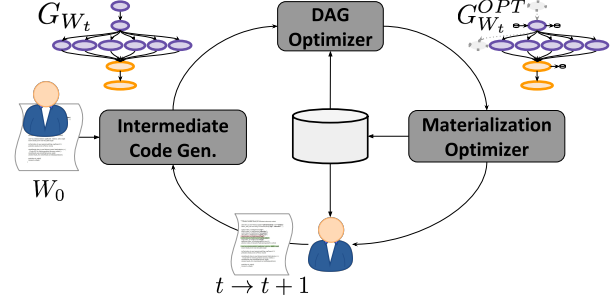
However, the majority of the development time is in fact spent *iterating on the machine learning workflow* by incrementally modifying steps within, including (i) *preprocessing*: altering data cleaning or extraction, or engineering features; (ii) *model training*: tweaking hyperparameters, or changing 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 [103], with the bulk of the time spent iterating on the machine learning workflow.

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 both difficult and error-prone. Due to the cumbersome and unreliable nature of this approach, developers often opt to rerun the entire workflow from scratch.

We present HELIX, a *declarative, general-purpose machine learning system that optimizes across iterations*. By optimizing across iterations, HELIX allows data scientists to avoid wasting time



(a) HELIX System architecture.



(b) Components in the HELIX workflow lifecycle.

Figure 3.1: HELIX architecture and workflow lifecycle.

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.

3.2 SYSTEM ARCHITECTURE

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 sub-tasks: data pre-processing (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 tasks in [Section 3.3](#). As we will demonstrate, these three tasks are generic and sufficient for describing a wide variety of supervised, semi-supervised, and unsupervised settings.

3.2.1 Components

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 3.1\(a\)](#). The three

components work collectively to *minimize the execution time for both the current iteration and subsequent iterations*:

1. Programming Interface (Section 3.3). HELIX provides a single Scala interface named `Workflow` for programming the entire workflow; the HELIX DSL also enables embedding of imperative code in declarative statements. Through just a handful of extensible operator types, the DSL supports a wide range of use cases for both data pre-processing and machine learning.

2. Compilation (Sections 3.4, 3.5.1 and 3.5.2). 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 (Section 3.4). 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 (Sections 3.5.1 and 3.5.2).

3. Execution Engine (Section 3.5.3). 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 [104] for data processing and domain-specific libraries such as CoreNLP [45] and Deeplearning4j [105] 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 Section 3.5.

3.2.2 The Workflow Lifecycle

Given the system components described in the previous section, Figure 3.1(b) 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:

- **DAG Compilation.** The `Workflow` W_t is compiled into a DAG G_{W_t} of operator outputs.
- **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.
- **Materialization Optimization.** During execution, the materialization optimizer determines which nodes in $G_{W_t}^{OPT}$ should be persisted to disk for future use.
- **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.

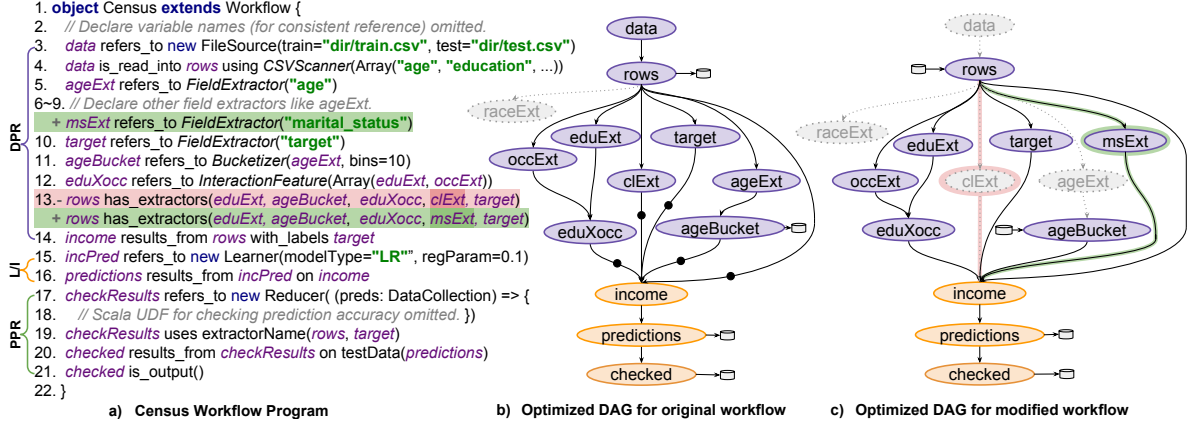


Figure 3.2: Example workflow for predicting income from census data.

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.

3.2.3 Example Workflow

We demonstrate the usage of HELIX with a simple example ML workflow for predicting income using census data from Kohavi [106], shown in Figure 3.2(a); this workflow will serve as a running example throughout the chapter. 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 3.2(a) into a DAG, which contains all nodes including `raceExt` and all edges (including the dashed edge) except the ones marked with dots in Figure 3.2(b). 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 very common in real machine learning development [107]. This example illustrates that

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

3.3 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 [108], 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 [109].

3.3.1 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 [43], one of the most comprehensive ML libraries, thereby demonstrating coverage. In Section 3.3.2, we introduce HML, which implements the capabilities offered by \mathcal{F} .

As mentioned in Section 3.2, 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 [42], TFX [47], 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 [110]. 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 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.

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}
eval: <code>score(y_{true}, y_{pred})</code>	<i>join</i> y_{true} and y_{pred} into a single dataset \mathcal{D} , then <i>reduce</i>
eval: <code>score(op, X, y)</code>	<i>inference</i> , then <i>join</i> , then <i>reduce</i>
selection: <code>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} .

Comparison with Scikit-learn

A dataset in Scikit-learn is represented as a matrix of FVs, denoted by \mathbf{X} . This is conceptually equivalent to $\mathcal{D} = \{\mathbf{x}^d\}$ introduced earlier, as the order of rows in \mathbf{X} is not relevant. Operations in Scikit-learn are categorized into dataset loading and transformations, learning, and model selection and evaluation [111]. 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 3.3.1, 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 [112]:

- **Estimator**, used to indicate that an operation has data-dependent behavior via a `fit(X[, y])` method, where \mathbf{X} contains FVs or raw records, and \mathbf{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 \mathbf{X}_{new} .

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.

3.3.2 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 3.2a). The complete grammar for HML in Backus-Naur Form as well as the semantics of all of the expressions can be found in Appendix B. Here, we describe high-level concepts including DCs and operators and discuss the strengths and limitations of HML in Section 3.3.3.

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 3.2(a), or from operations on other DCs, e.g., rows in Line 4 in Figure 3.2(a). 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 3.3.2, 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 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 3.2\(a\)](#). This design ensures that both training and test data undergo the exact same data pre-processing steps, eliminating bugs caused by inconsistent data pre-processing 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.

Operators

Operators in HELIX are designed to cover the functions enumerated in [Section 3.3.1](#), 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 3.3.1](#) 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_{SU} s. CSVScanner in Line 4 of [Figure 3.2\(a\)](#) 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 3.2\(a\)](#).

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 3.2(a) 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 3.2(a) Line 17 is a reducer that computes the prediction accuracy of the inference results in `predictions`.

3.3.3 Scope and Limitations

Coverage. In Section 3.3.1, 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 3.3.2 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 [42] and Weka [39], 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 Section 3.6 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 [33] and Pytorch [113], 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 3.3.2. 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. [114, 115], orthogonal to (and can therefore be combined with) our work, which operates at a coarser granularity.

3.4 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.

3.4.1 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 3.1. *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 3.2(b) shows the Workflow DAG for the program in Figure 3.2(a). 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 (•) in Figure 3.2(b). 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 3.5.4.

3.4.2 Tracking Changes

As described in Section 3.2.2, 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 3.2. 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 3.3. 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 [116]. The programming language community has a large body of work on verifying operational equivalence for specific classes of programs [117, 118, 119]. 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 3.1. At iteration $t + 1$, if an operator n_i^{t+1} is original, it must be recomputed.

With **Constraint 3.1**, our current approach to tracking changes yields the following guarantee on result correctness:

Theorem 3.1. 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 $t + 1$.

Proof. 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 3.1**, 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 3.1**. Therefore, the results for W_t are correct $\forall t \geq 0$.

3.5 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 3.5.2), and OPT-MAT-PLAN, which decides what to materialize to accelerate future iterations (Section 3.5.3). We finally discuss pruning optimizations to eliminate redundant computations (Section 3.5.4). We begin by introducing common notation and definitions.

3.5.1 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 3.3, 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 3.2. 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

$$T(W, s) = \sum_{n_i \in N} \mathbb{I}\{s(n_i) = S_c\} c_i + \mathbb{I}\{s(n_i) = S_l\} l_i \quad (3.1)$$

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

Clearly, setting all nodes to S_p trivially minimizes Equation (3.1). However, recall that **Constraint 3.1** 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 3.2** 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 (3.1) next.

3.5.2 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 3.1. (*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 3.1** and **Constraint 3.2**.*

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

$$T^*(W) = \min_s T(W, s) \quad (3.2)$$

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 3.2, 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 3.1**).

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 3.2** 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 the DAG on the left in Figure 3.3, 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 3.1** 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 [120].

Problem 3.2. *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*

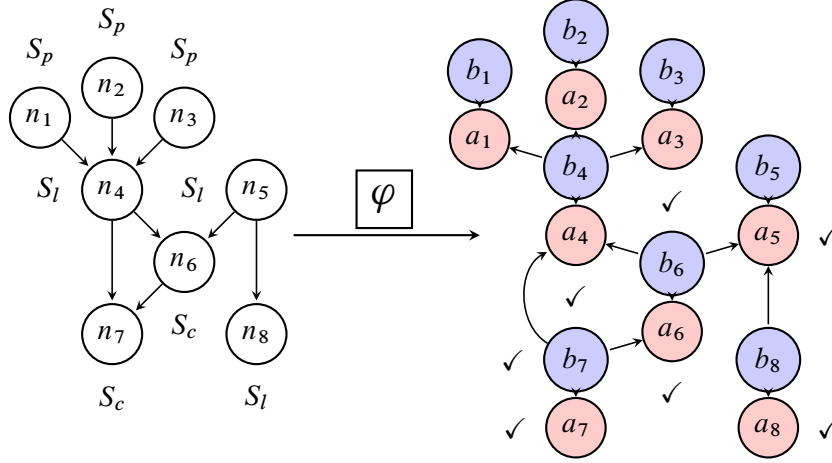


Figure 3.3: Transforming a Workflow DAG to a set of projects and dependencies. Checkmarks (✓) in the RHS DAG indicate a feasible solution to PSP, which maps onto the node states (S_p , S_c , S_l) in the LHS DAG.

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 3.1** 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 3.3** for an example instance of x , is shown in **Algorithm 3.1**. 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 3.3**. 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.

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 3.2**. For a project

Algorithm 3.1: 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  $(n_i, n_j) \in \{edges\ leaving\ from\ n_i\} \subseteq E$  do  
   // Add  $a_i$  as prerequisite for  $b_j$ .;  
9    $prerequisite[b_j] \leftarrow prerequisite[b_j] \cup a_i$ ;  
10  end  
11 end  
   //  $A$  is the set of projects selected by PSP;  
12  $A \leftarrow \text{PSP}(P, profit[], prerequisite[])$ ;  
13 for  $n_i \in N$  do // Map PSP solution to node states  
14   if  $a_i \in A$  and  $b_i \in A$  then  
15   |  $s[n_i] \leftarrow S_c$ ;  
16   end  
17   else if  $a_i \in A$  and  $b_i \notin A$  then  
18   |  $s[n_i] \leftarrow S_l$ ;  
19   end  
20   else  
21   |  $s[n_i] \leftarrow S_p$ ;  
22   end  
23 end  
24 return  $s[]$ ; // State assignments for nodes in  $G_W$ .
```

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 3.2. *Given an instance of OPT-EXEC-PLAN x , the reduction in Algorithm 3.1 produces a feasible and optimal solution to x .*

See Appendix A.1 for the proof.

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 [120]. We use the Edmonds-Karp algorithm [121] 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 3.4.2. 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 $a + b \neq b + a$.

3.5.3 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 Section 3.6), 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 3.4. 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

$$T_M(W_t) = \sum_{n_i \in M} l_i + T^*(W_{t+1}) \quad (3.3)$$

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 3.5.2](#), with M materialized.

[Equation \(3.3\)](#) defines the optimization objective for OMP.

Problem 3.3. (*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.,

$$\operatorname{argmin}_{M \subseteq N_t} \sum_{n_i \in M} l_i + T^*(W_{t+1}) \quad (3.4)$$

As discussed in [\[107\]](#), 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 3.3. *OPT-MAT-PLAN is NP-hard.*

See [Appendix A.2](#) for the proof.

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 3.5.2](#) 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 3.5. 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 3.3. 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 3.3](#).

Algorithm 3.2: 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        $S \leftarrow S - s_i$   
9     end  
10  end  
11 end
```

Definition 3.6. Given Workflow DAG $G_w = (N, E)$, the cumulative run time for a node n_i is defined as

$$C(n_i) = t(n_i) + \sum_{n_j \in \text{ancestors}(n_i)} t(n_j) \quad (3.5)$$

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

Algorithm 3.2 shows the heuristics employed by HELIX’s materialization optimizer to decide what intermediate results to materialize. In essence, Algorithm 3.2 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 3.2 would have materialized n_i previously, allowing us to make the right choice for n_i . Otherwise, Algorithm 3.2 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 simply loading n_i .

Due to the streaming Constraint 3.3, complex dependencies between descendants of ancestors such as the one between n_5 and n_8 in Figure 3.3 previously described in Section 3.5.2, are ignored by Algorithm 3.2—we cannot retroactively update our decision for n_5 after n_8 has been run. We show in Section 3.6 that this simple algorithm is effective in multiple application domains.

Limitations of Streaming OMP. The streaming OMP heuristic given in [Algorithm 3.2](#) 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 3.2](#) 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 3.2](#) 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 3.3](#)) where the input is divided into mini batches processed end-to-end independently, [Algorithm 3.2](#) 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.

3.5.4 Workflow DAG Pruning

In addition to reusing intermediate result, 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 B.1](#) (see [Appendix B](#)). An example of an Extractor pruned in this fashion is `raceExt` (grayed out) in [Figure 3.2\(b\)](#), 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

	Census [122]	Genomics [123]	IE [10]	MNIST [8]
Num. Data Source	Single	Multiple	Multiple	Single
Input to Ex. Mapping	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	Classification	Unsupervised	Structured Pred.	Classification
Application Domain	Social Sciences	Natural Sciences	NLP	Computer Vision
Supp. by HELIX	✓	✓	✓	✓
Supp. by KeystoneML	✓	✓		✓*
Supp. by DeepDive	✓*		✓*	

Table 3.2: 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.

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 B.1, 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.

3.6 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.

3.6.1 Systems and Baselines for Comparison

We compare the optimized version of HELIX, HELIX OPT, against two state-of-the-art ML workflow systems: KeystoneML [8], and DeepDive [9]. 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 [8] 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 [9, 10] 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 3.1 for the optimal reuse strategy and Algorithm 3.2 to decide what to materialize.

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

3.6.2 Workflows

We conduct our experiments using four real-world ML workflows spanning a range of application domains. Section 3.6 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 pre-processing 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 pre-processing 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 [122]. It uses the Census Income dataset [124], with 14 attributes representing demographic information, with the goal to predict whether a person’s annual income is >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 3.2(a). This workflow evaluates a system’s efficiency in handling simple ML tasks with fine-grained feature engineering.

Genomics Workflow. This workflow involves 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 [125]; 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 pre-processing 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 [10]. 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 pre-processing.

MNIST Workflow. The MNIST dataset [126] contains images of handwritten digits to be classified, which is a well-studied task in the computer vision community, from the KeystoneML [8] evaluation. The workflow involves nondeterministic (and hence not reusable) data pre-processing, 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 ✓* in [Section 3.6](#) 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 3.6.5](#).

3.6.3 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 [107] 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).

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.

3.6.4 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.

3.6.5 Evaluation vs. State-of-the-art Systems

Use Case Support

HELIX supports ML workflows in multiple distinct application domains, spanning tasks with varying complexity in both supervised and unsupervised learning.

Recall that the four workflows used in our experiments are in social sciences, NLP, computer vision, and natural sciences, respectively. [Section 3.6](#) 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 pre-processing. On the other hand, DeepDive is highly specialized for information extraction and focuses on supporting data pre-processing. Unfortunately, its learning and evaluation components are not configurable by the user, limiting the

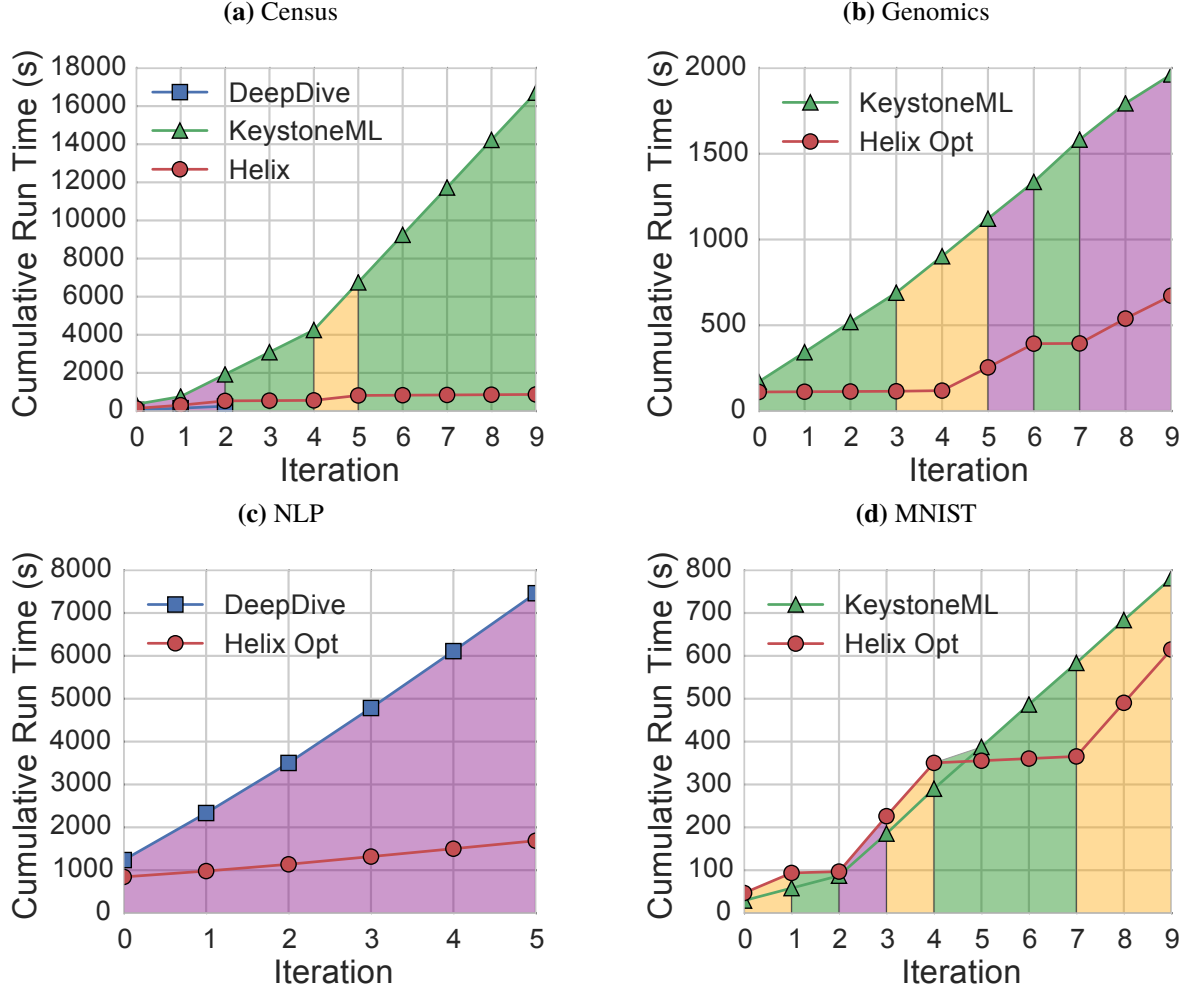


Figure 3.4: Cumulative run time for the four workflows. The color under the curve indicates the type of change in each iteration: purple for DPR, orange for L/I, and green for PPR.

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.

Cumulative Run Time

HELIX achieves up to **19** \times cumulative run time reduction in ten iterations over state-of-the-art ML systems.

Figure 3.4 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

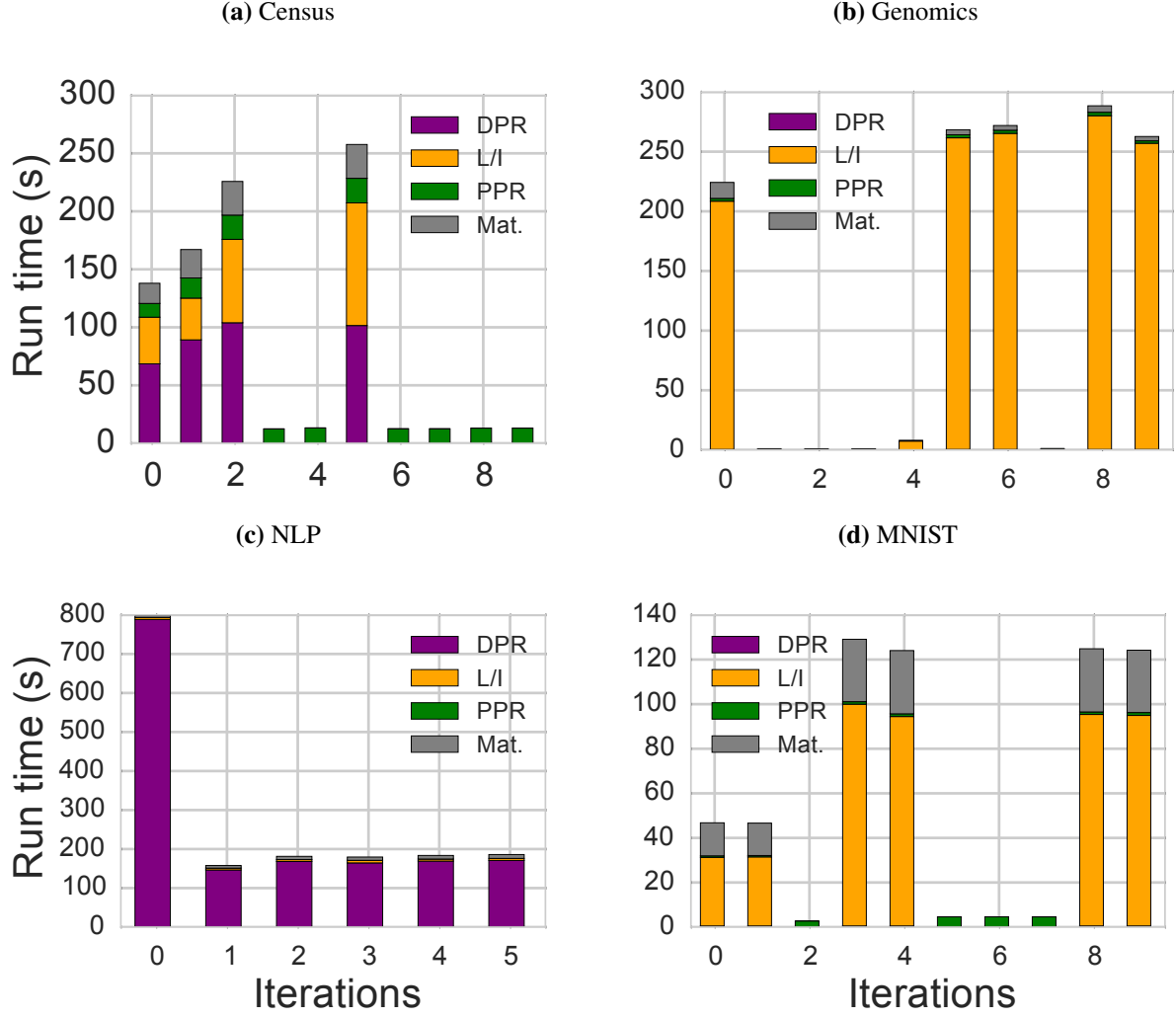


Figure 3.5: HELIX per-iteration run time breakdown by workflow component.

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 3.5** shows the breakdown by workflow components and materialization for the individual iteration run times in HELIX, with the same color scheme as in **Figure 3.4** for the workflow components and gray for materialization time.

Census. As shown in **Figure 3.4(a)**, the census workflow has the largest cumulative run time gap between HELIX OPT and the competitor systems—HELIX OPT is **19× 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 3.5(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 pre-processing 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 [107].

Genomics. In Figure 3.4(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 3.5(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 3.4(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 3.5(c), leading to large run time reductions in iterations 1-5 and thus a large cumulative run time reduction.

MNIST. Figure 3.4(d) shows the cumulative run times for the MNIST workflow. As mentioned above, the MNIST workflow has nondeterministic data pre-processing, 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 3.5(d), we see 1) DPR operations take negligible time, and HELIX OPT avoids wasteful materialization of their

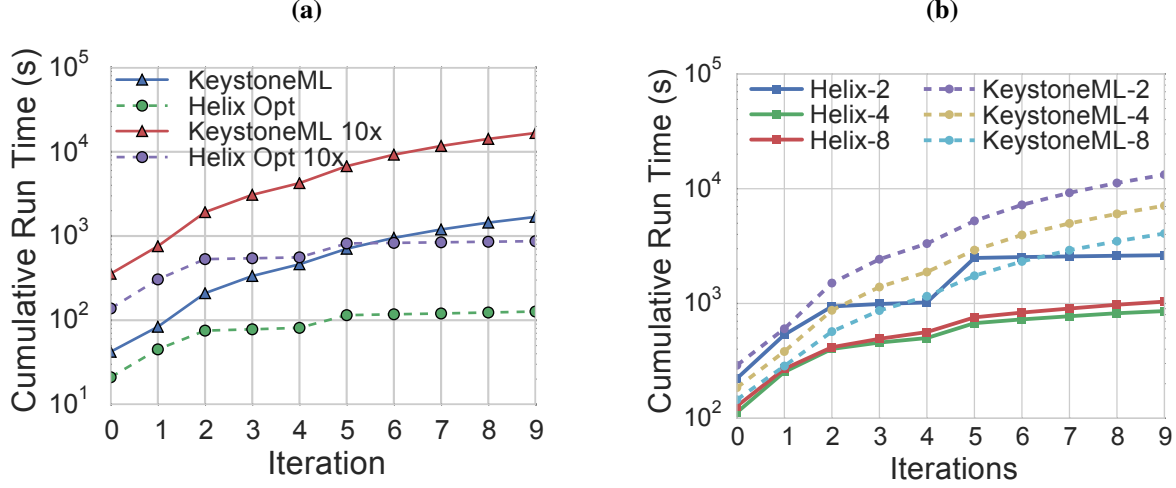


Figure 3.6: a) Census and Census 10x cumulative run time for HELIX and KeystoneML on a single node; b) Census 10x cumulative run time for HELIX and KeystoneML on different size clusters.

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.

Scalability vs. KeystoneML

Dataset Size. We test scalability of HELIX and KeystoneML with respect to dataset size by running the ten iterations in Figure 3.4(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 3.6(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 3.4(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 3.6(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 Section 3.3 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

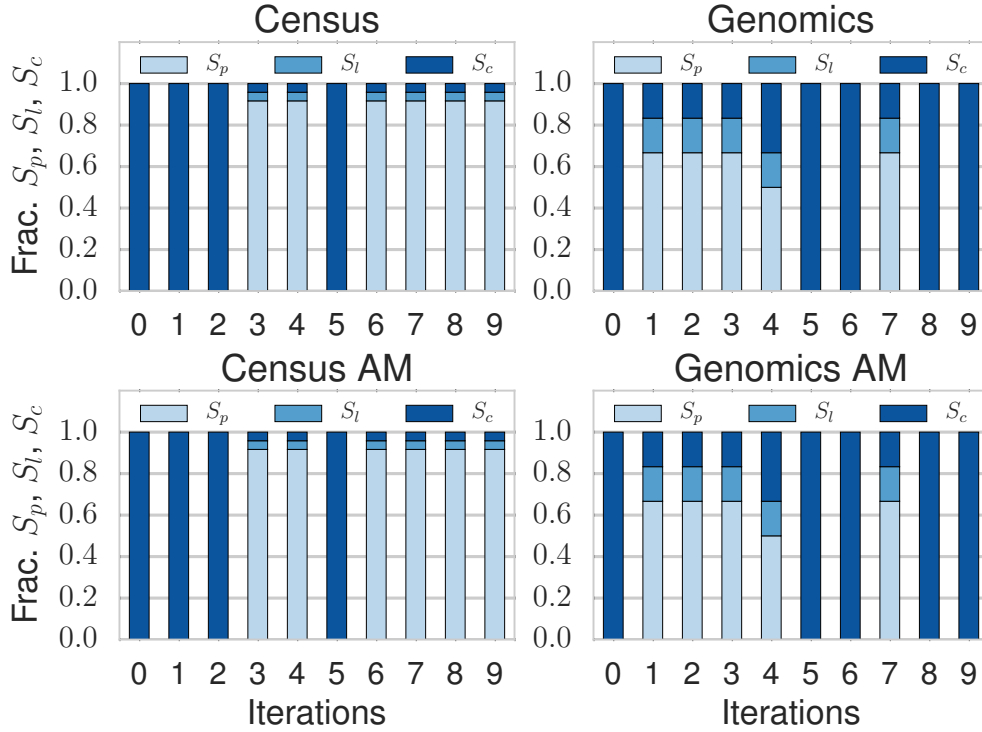


Figure 3.7: Fraction of states in S_p , S_l , S_c as determined by Algorithm 3.1 for the Census and Genomics workflows for HELIX OPT and HELIX AM.

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).

3.6.6 Evaluation vs. Simpler HELIX Versions

HELIX OPT achieves the lowest cumulative run time on all workflows compared to simpler versions of HELIX. HELIX AM often uses more than 30× the storage of HELIX OPT when able to complete in a reasonable time, while not being able to complete within 50× of the time taken by HELIX OPT elsewhere. HELIX NM takes up to 4× the time taken by HELIX OPT.

Next, we evaluate the effectiveness of Algorithm 3.2 at approximating the solution to the NP-hard OPT-MAT-PLAN problem. We compare HELIX OPT that runs Algorithm 3.2 against: HELIX AM that replaces Algorithm 3.2 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

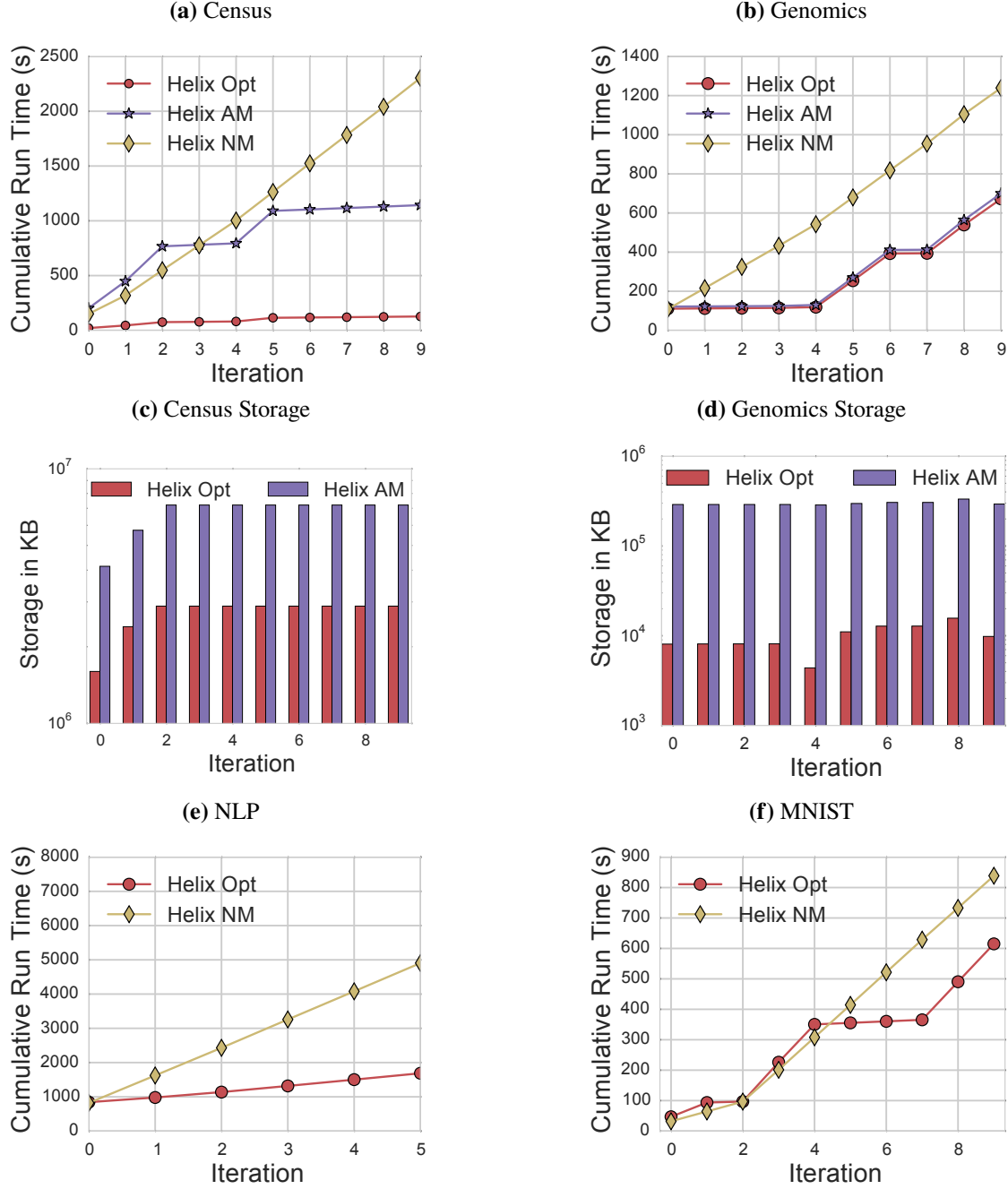


Figure 3.8: Cumulative run time and storage use against materialization heuristics on the same four workflows as in Figure 3.4.

most flexible choices for reuse. On the other hand, HELIX NM has no materialization time overhead but also offers no reuse.

Figures 3.8(a), 3.8(b), 3.8(e) and 3.8(f) show the cumulative run time on the same four workflows as in Figure 3.4 for the three variants.

HELIX AM is absent from [Figures 3.8\(e\) and 3.8\(f\)](#) 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 3.8\(e\) and 3.8\(f\)](#) 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 3.8\(a\) and 3.8\(b\)](#) 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 3.8\(c\) and 3.8\(d\)](#) 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 3.8\(c\)](#) and $\frac{1}{30}$ the storage of HELIX AM for the genomics workflow in [Figure 3.8\(d\)](#). 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 3.2](#), 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 3.7](#) 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.

3.6.7 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 3.9](#) 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, for iterations during which 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.

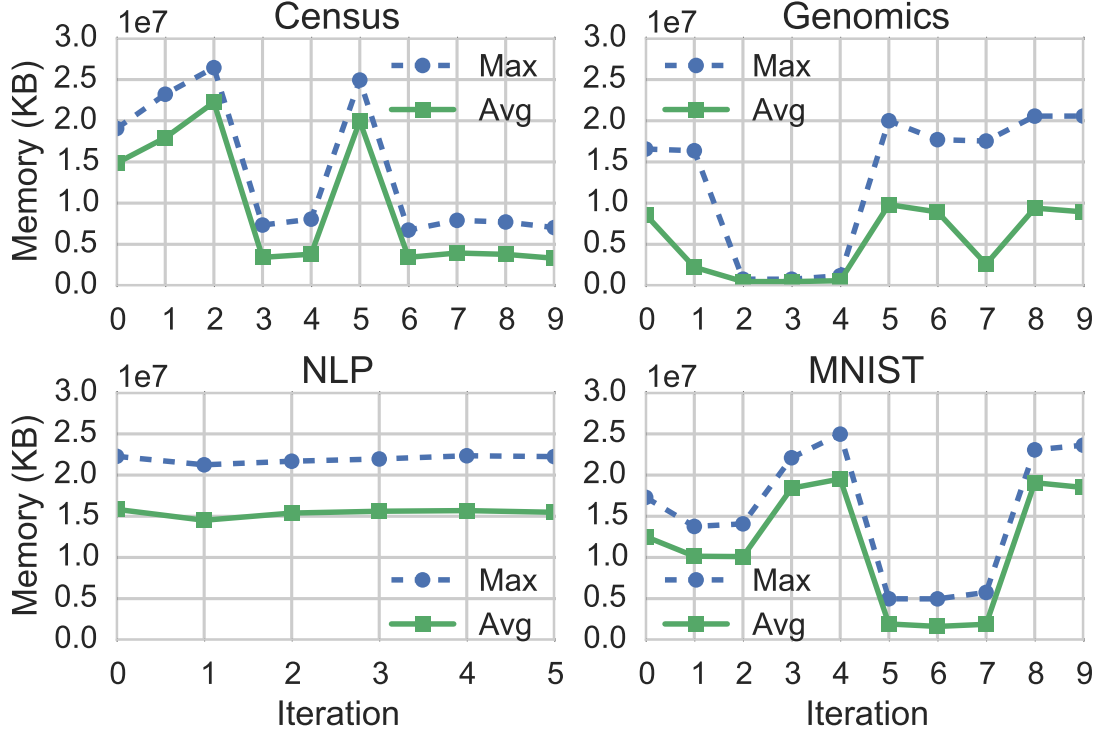


Figure 3.9: Peak and average memory for HELIX.

3.7 SUMMARY

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 chapter are general-purpose; other systems can enjoy the benefits of HELIX’s optimization modules through simple wrappers and connectors.

HELIX demonstrates how to leverage similarities in adjacent workflow iterations for safe and interactive ML model development; in the next chapter, we explore techniques that have downstream applications for safe and interactive browsing.

CHAPTER 4: SAFE BROWSING WITH LEARNED BLOOM FILTERS

In this chapter, we describe techniques for learning compact Bloom filters of multidimensional data. We begin by reviewing both traditional and learned Bloom filters, and then describe several techniques for further compressing learned Bloom filters, especially considering filters over multidimensional data. These compressed Bloom filters can be used for downstream browsing tasks, e.g., to fetch blocks that contain records satisfying some desired criteria. They obey similar semantics to traditional Bloom filters (guaranteeing absence of false negatives) while improving interactivity of the downstream browsing task thanks to their potentially significant space savings.

4.1 OVERVIEW

Bloom filters are widely used as existence indices in many applications [127]. By allowing some false positives, their space requirements depend only on the number of inserted keys (and are independent of the sizes of the individual keys). Despite their advantages, Bloom filters are unable to leverage differences between the set of in-index keys and the distribution of negative queries, and recent work [128] has proposed *learned Bloom filters* to model latent separations between the two. We expand on these ideas and show that learned filters are effective for indexing multidimensional data (k -tuples) when there exists some co-occurrence structure between in-index k -tuples that are different from out-of-index tuples. We explore a new kind of multidimensional Bloom index that exploits this structure, and our approach can offer significant space savings over traditional Bloom filters. We also give new methods to quantify the space savings that properly capture the asymmetry intrinsic to Bloom filters (i.e., allowing false positives, but not false negatives).

Multidimensional Data. Bloom filters are commonly used to index data with multiple attributes. Furthermore, any application using multiple Bloom filters to index different data into semantically different sets can equivalently use a single, monolithic filter over 2-dimensional data: instead of testing whether Bloom filter i contains item x , we check whether our large filter contains (i, x) . For a concrete example, the Windows implementation of the Git SCM uses Bloom filters to support filtered searches over the Git commit graph [129]. Specifically, each commit is associated with a Bloom filter into which all the file names for files added, deleted, or otherwise modified in the commit are inserted. Such an application is semantically equivalent to using a single filter over two dimensions – instead of querying the Bloom filter for commit `ad7ad3b` for file `file.txt`, one would instead query a single monolithic filter for the 2-tuple `(ad7ad3b, file.txt)`.

We observe that querying over subsets of attributes is often important in many applications; going with our Git example, if we want to check whether any commit modified file name, we can do so by

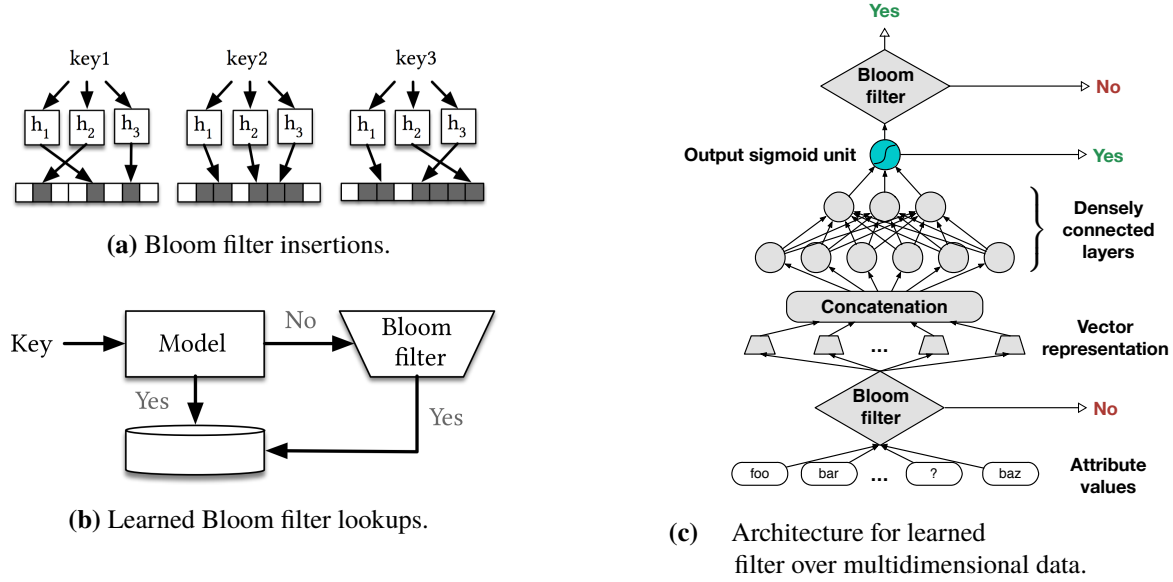


Figure 4.1: Illustration of traditional and learned Bloom filters.

querying for $(?, \text{name})$ where “?” represents a placeholder that can be matched by any value. In this work, we thus allow *wildcard queries*, in which only a subset of tuple entries are specified. To facilitate such queries, traditional Bloom filters must index tuples like $(a, b, ?)$ as first-class items.

We posit (and demonstrate) that learned filters can be much more space-efficient when allowing wildcard queries, since we observe in practice that co-occurrence structure is present even when only considering subsets of attributes. Even before taking into account all of the extra keys resulting from introduction of wildcards, the number of distinct combinations of attribute values tends to grow exponentially as we increase the number of indexed dimensions. Learned filters have the ability to combat this by leveraging the co-occurrence structure present in the in-index data.

4.2 LEARNING FILTERS

Background. The idea behind learned Bloom filters, as originally described in [128], is illustrated in Figures 4.1(a) and 4.1(b). Bloom filter insertions are facilitated by setting h bits in a bit array, whose positions correspond to the output of h independent random hash functions. To determine whether a key is considered in-index, all h hash function outputs must hash to set bits. In this way, false positives are possible, but false negatives are not. Learned Bloom filters model Bloom filter lookups as a classification problem. To train a learned Bloom filter, in-index items, \mathcal{K} , are used as positive training examples, and negative training examples are drawn from a negative query distribution $\mathcal{D}_{\overline{\mathcal{K}}}$. Because the model may have false negatives, the set of all false negatives

are inserted into a post-filter (“fixup” filter) that is checked whenever the model gives a negative prediction. Concretely, for a classifier f with prediction threshold τ , all $k \in \mathcal{K}$ for which $f(k) < \tau$ (the set of which is denoted as $\mathcal{K}_{<\tau}$) are inserted into the spillover filter. This backup filter is then checked whenever f gives an output below τ in order to eliminate the possibility of false negatives. For a classifier with false positive rate $F_p^{(1)}$ and a backup traditional Bloom filter with false positive rate $F_p^{(2)}$, the overall false positive rate of the learned Bloom filter is given by $F_p^{(1)} + (1 - F_p^{(1)})F_p^{(2)}$. When $F_p^{(1)}$ is small, this is approximately $\approx F_p^{(1)} + F_p^{(2)}$. For a detailed discussion of how classifier performance relates to learned Bloom filter size, please see [128] or [63].

Modeling Multidimensional Data. Figure 4.1(c) depicts our model architecture for handling multidimensional inputs. In this work, we consider k -tuples of strings. Each string attribute value is first converted into an embedding vector (we defer specific discussion to the next section). Furthermore, each “wildcard” placeholder is assigned a separate embedding vector per dimension. To model any co-occurrence structure between the attribute values, these vectors are concatenated and fed through a dense layer, whose output is then converted to a logit by a sigmoid head. We train the DNN and the embeddings using Adam [130] on positive examples that represent the in-index elements and negative examples synthesized from queries for out-of-index data.

4.3 CHALLENGES AND OPTIMIZATIONS

We identify three key challenges when learning Bloom filters over multidimensional data:

- **Challenge 1: Tradeoff between model size and model performance.**
- **Challenge 2: Suboptimal performance at low false positive rates.**
- **Challenge 3: Noisy in-index data obscures overall co-occurrence patterns.**

We introduce three key optimizations: (1) *modeling high-cardinality attributes with RNNs*, (2) *leveraging the sandwiching technique introduced in [64]*, and (3) *robust learning via ℓ_1 -regularized shift parameters* in order to address each of these respective challenges. We now give a detailed description of each optimization.

Optimization 1: Model high-cardinality attributes with RNNs. Using direct embedding lookups for high-cardinality attributes creates many model parameters, and in the worst case results in learned Bloom filters that greatly exceed traditional Bloom filters in size. To cope with this, we convert attribute values for high-cardinality attributes (say, > 500 distinct values) into vectors using character-level recurrent neural networks (RNNs) with gated recurrent units (GRUs) [131]. We opt for direct embeddings for the low-cardinality attributes when converting their corresponding attribute values to vectorized form, as the benefit in learned Bloom filter size due to the improvement

in the quality of the model predictions outweighs the cost of extra space used due to the increase in model parameters (compared to RNNs).

Optimization 2: Choose better classifier cutoffs with sandwiching. To enforce a low target false positive rate FPR , it can be necessary to choose a very large classifier cutoff τ . Although there exist strategies for improving performance at low false positive rates, we opt to sidestep the issue altogether by employing a strategy for learned Bloom filters called *sandwiching* [64]. In brief, this strategy introduces an additional traditional Bloom filter in the learned Bloom filter architecture depicted in Figure 4.1(b) as a *prefilter*, into which all in-index keys in \mathcal{K} are inserted. Thus, a query must be found in the prefilter to be considered positive; otherwise, it is immediately known to be out-of-index. Although all of \mathcal{K} is inserted into the prefilter, it need not occupy many bits as it can be very porous, having a very high false positive rate compared to the target false positive rate FPR .

When leveraging sandwiching, we are free to choose the classifier cutoff τ for the model portion of the learned Bloom filter however we wish. A natural question, then, is how to choose this threshold. The following theorem shows that the best cutoff for a sandwiched filter with nonempty prefilter maximizes the KL divergence between the classifier’s true positive and false positive rates.

Theorem 4.1. *Suppose one has surrounded two learned models with traditional Bloom filters to create sandwiched Bloom filters (with same FPR), and further assume that each sandwiched filter uses the optimal number of bits and leverages a nonempty prefilter to do so. Let X_{T_p} and X_{F_p} be Bernoulli random variables with $\mathbb{E}[X_{T_p}] = T_p$ and $\mathbb{E}[X_{F_p}] = F_p$, where T_p is the fraction of in-index items correctly classified by the learned model, and F_p is the learned model’s false positive rate (expected proportion of out-of-index queries predicted as in-index). Then the sandwiched filter with the higher value of $KL(X_{T_p}||X_{F_p})$ uses fewer bits for the traditional prefilter and postfilter components.*

Please see the appendix (§A.3) for the proof.

Optimization 3: Robust learning with ℓ_1 -regularized shift parameters. Our final optimization attempts to address noise present in the in-index data. To do so, we borrow an idea from robust learning, whereby each positive training example k is associated with a *shift parameter* γ_k . The vector of shift parameters $\vec{\gamma}$ is ℓ_1 -regularized to encourage sparsity. Although originally introduced in the context of linear regression [132] and logistic regression [133], we found that the technique works well in deep models too. In brief, the output of the sigmoid head $g(z_k) = \frac{1}{1+\exp(-z_k)}$ is shifted as $g(z_k + \gamma_k) = \frac{1}{1+\exp(-z_k - \gamma_k)}$. In our experiments on a dataset of airline flights [134], this technique reduced the number of bits required for a learned Bloom filter by 15%.

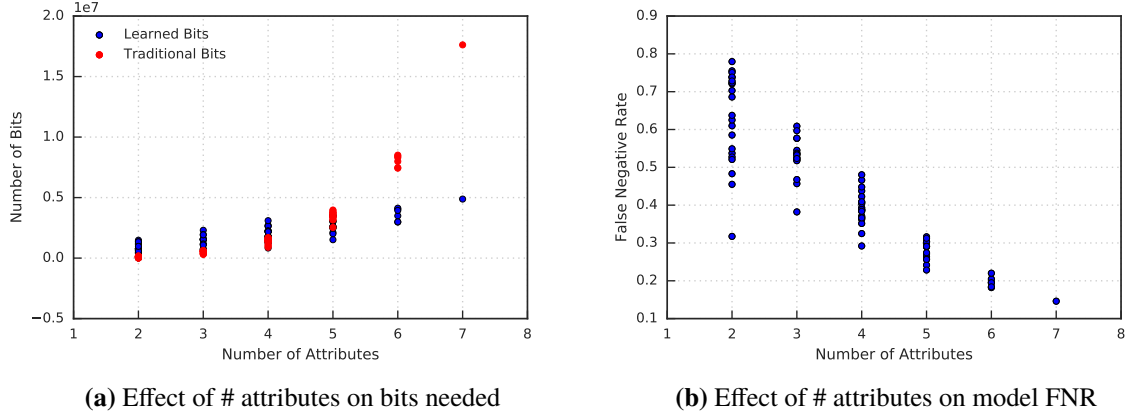


Figure 4.2: Effect of varying number of attributes indexed. Each point corresponds to a learned Bloom filter that achieves 1% FPR for a combination of indexed attributes, across all subsets of two or more indexed attributes.

4.4 EXPERIMENTAL RESULTS

Production Recommender System Logs. We consider a dataset of 70000 subsampled impressions from the logs of a major production recommendation system, where each impression consists of 7 attributes. Although the number of rows (i.e. impressions) is relatively small, by allowing wildcard-style queries, we observe 1.8 million distinct co-occurring attribute values, all of which must be indexed. Since the false positive rate of learned Bloom filters is sensitive to the distribution of incoming negative queries [63, 128], for each set of indexed attributes, we synthesized an out-of-index query distribution. This distribution was generated by joining random pairs of non co-occurring observed attribute values, uniformly across pairs of indexed attributes (queries with unobserved attribute values will be eliminated during embedding lookups, with high probability), although one could also use query logs if available. Figure 4.2 shows the result of indexing every subset of attributes, using both a traditional filter and a learned filter. Without RNNs (not shown), the learned model uses more bits than a traditional filter. Leveraging sandwiching reduces the number of bits needed by about 10%. Overall, we achieve greater space savings with more indexed attributes, achieving space reductions of **72%** over traditional Bloom filters when indexing all 7 attributes.

Flights. We divided 7 million rows from year 2008 into contiguous pages (following the order in which we obtained the data) of 10000 rows each, and index all pairs of (Airport, Page ID). We join known airports with random page IDs for a query negative distribution, achieve space reductions of **25%** over traditional bloom filters. By incorporating ℓ_1 -regularized shift parameters, we increase the space reductions to **35%**.

4.5 SUMMARY

Based on our experimental findings, learned Bloom filters show promising performance when indexing multidimensional data while allowing flexible queries with wildcards. Each of our key optimizations: *separate modeling of high-cardinality attributes with RNNs*, *sandwiching with optimal classifier cutoffs*, and *robust learning with shift parameters* addresses a key issue when learning Bloom filters over multidimensional data. The additional space savings can then improve interactivity of downstream browsing tasks.

The next two chapters develop systems for interactive analytics. We first consider data-aware techniques for visual exploration in [Chapter 5](#), and then extend these techniques to more general report generation in [Chapter 6](#).

CHAPTER 5: SAFE APPROXIMATION FOR VISUAL EXPLORATION

In this chapter, we describe FASTMATCH, a system that uses sampling-based AQP with guarantees to find histograms that match a desired target, thereby helping enable safe and interactive visual exploration. We begin by motivating and formulating the problem, then describe our core algorithmic and architectural techniques, and finally use several real datasets to evaluate FASTMATCH along the axes of safety and interactivity.

5.1 MOTIVATION

In exploratory data analysis, analysts often generate and peruse a large number of visualizations to identify those that *match desired criteria*. This process of iterative “generate and test” occupies a large part of visual data analysis [16, 17, 18], and is often cumbersome and time consuming, especially on very large datasets that are increasingly the norm. This process ends up impeding interaction, preventing exploration, and delaying the extraction of insights.

Example 1: Census Data Exploration. Alice is exploring a census dataset consisting of hundreds of millions of tuples, with attributes such as gender, occupation, nationality, ethnicity, religion, adjusted income, net assets, and so on. In particular, she is interested in understanding how applying various filters impacts the relative distribution of tuples with different attribute values. She might ask questions like *Q1*: Which countries have similar distributions of wealth to that of Greece? *Q2*: In the United States, which professions have an ethnicity distribution similar to the profession of doctor? *Q3*: Which (nationality, religion) pairs have a similar distribution of number of children to Christian families in France?

Example 2: Taxi Data Exploration. Bob is exploring the distribution of taxi trip times originating from various locations around Manhattan. Specifically, he plots a histogram showing the distribution of taxi pickup times for trips originating from various locations. As he varies the location, he examines how the histogram changes, and he notices that choosing the location of a popular nightclub skews the distribution of pickup times heavily in the range of 3am to 5am. He wonders *Q4*: Where are the other locations around Manhattan that have similar distributions of pickup times? *Q5*: Do they all have nightclubs, or are there different reasons for the late-night pickups?

Example 3: Sales Data Exploration. Carol has the complete history of all sales at a large online shopping website. Since users must enter birthdays in order to create accounts, she is able to plot the age distribution of purchasers for any given product. To enhance the website’s recommendation engine, she is considering recommending products with similar purchaser age distributions. To test the merit of this idea, she first wishes to perform a series of queries of the form *Q6*: Which

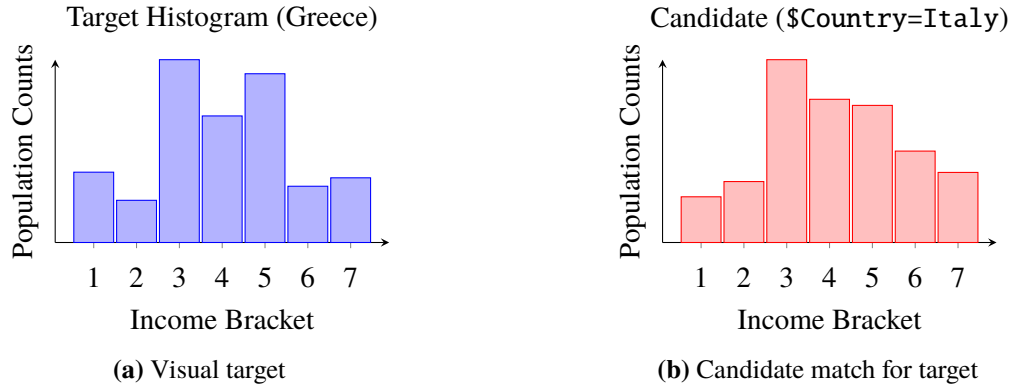


Figure 5.1: Example visual target and candidate histogram

products were purchased by users with ages most closely following the distribution for a certain product—a particular brand of shoes, or a particular book, for example? Carol wishes to perform this query for a few test products before integrating this feature into the recommendation pipeline. These cases represent scenarios that often arise in exploratory data analysis—finding matches to a specific distribution. The focus of this chapter is to *develop techniques for rapidly exploring a large class of histograms to find those that match a user-specified target*.

Referring to *Q1* in the first example, a typical workflow used by Alice may be the following: first, pick a country. Generate the corresponding histogram. This could be done either using a language like R, Python, or Javascript, with the visualization generated in ggplot [135] or D3 [136], or using interactions in a visualization platform like Tableau [137]. Does the visualization look similar to that of Greece? If not, pick another, generate it, and repeat. Else, record it, pick another, generate it, and repeat. If only a select few countries have similar distributions, she may spend a huge amount of time sifting through her data, or may simply give up early.

5.2 PROBLEM FORMULATION

In this section, we formalize the problem of identifying histograms whose distributions match a reference.

5.2.1 Generation of Histograms

We start with a concrete example of the typical database query an analyst might use to generate a histogram. Returning to our example from Section 5.1, suppose an analyst is interested in studying how population proportions vary across income brackets for various countries around the world. Suppose she wishes to find countries with populations distributed across different income brackets

Symbol(s)	Description
X, Z, V_X, V_Z, T	x-axis attribute, candidate attribute, respective value sets, and relation over these attributes, used in histogram-generating queries (see Definition 5.1)
$k, \delta, \varepsilon, \sigma$	User-supplied parameters (number of matching histograms to retrieve, error probability upper bound, approximation error upper bound, selectivity threshold (below which candidates may optionally be ignored))
$\mathbf{q}, \mathbf{r}_i, \mathbf{r}_i^*, (\bar{\mathbf{q}}, \bar{\mathbf{r}}_i, \bar{\mathbf{r}}_i^*)$	Visual target, candidate i 's estimated (unstarred) and true (starred) histogram counts (normalized variants)
$d(\cdot, \cdot)$	Distance function, used to quantify visual distance (see Definition 5.2)
$n_i, n'_i, \varepsilon_i, \delta_i, \tau_i (\tau_i^*)$	Quantities specific to candidate i during HistSim run: number of samples taken, estimated samples needed (see Section 5.4), deviation bound (see Definition 5.4), confidence upper bound on ε_i -deviation or rareness, and distance estimate from \mathbf{q} (true distance from \mathbf{q}), respectively
$n_i^\partial, \mathbf{r}_i^\partial, \tau_i^\partial$	Quantities corresponding to samples taken in a specific round of HistSim stage 2: number of samples taken for candidate i in round, per-group counts for candidate i for samples taken in round, corresponding distance estimates using the samples taken in round, respectively
M, A	Set of matching histograms (see Definition 5.3) and non-pruned histograms, respectively, during a run of HistSim
$N_i, N, m, f(\cdot; N, N_i, m)$	Number of datapoints corresponding to candidate i , total number of datapoints, samples taken during stage 1, hypergeometric pdf

Table 5.1: Summary of notation used in [Chapter 5](#).

most similarly to a specific country, such as Greece. Consider the following SQL query, where \$COUNTRY is a variable:

```
SELECT income_bracket, COUNT(*)
FROM census
WHERE country=$COUNTRY
GROUP BY income_bracket
```

Figure 5.2: SQL for computing income by country

This query returns a list of 7 (income bracket, count) pairs to the analyst for a specific country. The analyst may then choose to visualize the results by plotting the counts versus different income brackets in a histogram, i.e., a plot similar to the right side of [Figure 5.1](#) (for Italy). Currently, the analyst may examine hundreds of similar histograms, one for each country, comparing it to the one for Greece, to manually identify ones that are similar.

In contrast, the goal of FASTMATCH is to perform this search automatically and efficiently. Conceptually, FASTMATCH will iterate over all possible values of country, generate the corresponding histograms, and evaluate the similarity of its distribution (based on some notion of similarity described subsequently) to the corresponding visualization for Greece. In actuality, FASTMATCH

will perform this search all at once, quickly pruning countries that are either clearly close or far from the target.

Candidate Visualizations. Formally, we consider visualizations as being generated as a result of **histogram-generating queries**:

Definition 5.1. A histogram-generating query is a SQL query of the following type: The table T

```
SELECT X, COUNT(*) FROM T
WHERE Z=z_i GROUP BY X
```

Figure 5.3: Template for histogram-generating query

and attributes X and Z form the query’s template.

For each concrete value z_i of attribute Z specified in the query, the results of the query—i.e., the grouped counts—can be represented in the form of a vector (r_1, r_2, \dots, r_n) , where $n = |V_X|$, the cardinality of the value set of attribute X . This n -tuple can then be used to plot a histogram visualization—in this chapter, when we refer to a histogram or a visualization, we will be typically referring to such an n -tuple. For a given *grouping attribute* X and a *candidate attribute* Z , we refer to the set of all visualizations generated by letting Z vary over its value set as the set of **candidate visualizations**. We refer to each distinct value in the grouping attribute X ’s value set as a *group*. In our example, X corresponds to `income_bracket` and Z corresponds to `country`.

For ease of exposition, we focus on candidate visualizations generated from queries according to **Definition 5.1**, having single categorical attributes for X and Z . Our methods are more general and extend naturally to handle (i) *predicates*: additional predicates on other attributes, (ii) *multiple and complex Xs*: additional grouping (i.e., X) attributes, groups derived from binning real-values (as opposed to categorical X), along with groups derived from binning multiple categorical X attribute values together (e.g., quarters instead of individual months), and (iii) *multiple and complex Zs*: additional candidate (i.e., Z) attributes, as well as candidate attribute values derived from binning real values (as opposed to categorical Z). The flexibility in specifying histogram-generating queries—exponential in the number of attributes—makes it impossible for us to precompute the results of all such queries.

Visualization Terminology. Our methods are agnostic to the particular method used to present visualizations. That is, analysts may choose to present the results generated from queries of the form in **Definition 5.1** via line plots, heat maps, choropleths, and other visualization types, as any of these may be specified by an ordered tuple of real values and are thus permitted under our notion of a “candidate visualization”. We focus on bar charts of frequency counts and histograms—these naturally capture aggregations over the categorical or binned quantitative grouping attribute X respectively. Although a bar graph plot of frequency counts over a categorical grouping attribute is

not technically a histogram, which implies that the grouping attribute is continuous, we loosely use the term “histogram” to refer to both cases in a unified way.

Visual Target Specification. Given our specification of candidate visualizations, a *visual target* is an n -tuple, denoted by \mathbf{q} with entries Q_1, Q_2, \dots, Q_n , that we need to match the candidates with. Returning to our flight delays example, \mathbf{q} would refer to the visualization corresponding to Greece, with Q_1 being the count of individuals in the first income bracket, Q_2 the count of individuals in the second income bracket, and so on.

Samples. To estimate these candidate visualizations, we need to take *samples*. In particular, for a given candidate i for some attribute Z , a sample corresponds to a single tuple t with attribute value $Z = z_i$. The attribute value $X = x_j$ of t increments the j th entry of the estimate \mathbf{r}_i for the candidate histogram.

Candidate Similarity. Given a set of candidate visualizations with estimated vector representations $\{\mathbf{r}_i\}$ such that the i th candidate is generated by selecting on $Z = z_i$, our problem hinges on finding the candidate whose distribution is most “similar” to the visual target \mathbf{q} specified by the analyst. For quantifying visual similarity, we do not care about the absolute counts $r_1, r_2, \dots, r_{|V_X|}$, and instead prefer to determine whether \mathbf{r}_i and \mathbf{q} are close in a *distributional* sense. Using hats to denote normalized variants of \mathbf{r}_i and \mathbf{q} , write

$$\bar{\mathbf{r}}_i = \frac{\mathbf{r}_i}{\mathbf{1}^T \mathbf{r}_i} \quad \bar{\mathbf{q}} = \frac{\mathbf{q}}{\mathbf{1}^T \mathbf{q}} \quad (5.1)$$

With this notational convenience, we make our notion of similarity explicit by defining candidate distance as follows:

Definition 5.2. For candidate \mathbf{r}_i and visual predicate \mathbf{q} , the *distance* $d(\mathbf{r}_i, \mathbf{q})$ between \mathbf{r}_i and \mathbf{q} is defined as follows:

$$d(\mathbf{r}_i, \mathbf{q}) = \|\bar{\mathbf{r}}_i - \bar{\mathbf{q}}\|_1 = \left\| \frac{\mathbf{r}_i}{\mathbf{1}^T \mathbf{r}_i} - \frac{\mathbf{q}}{\mathbf{1}^T \mathbf{q}} \right\|_1 \quad (5.2)$$

That is, after normalizing the candidate and target vectors so that their respective components sum to 1 (and therefore correspond to distributions), we take the ℓ_1 distance between the two vectors. When the target \mathbf{q} is understood from context, we denote the distance between candidate \mathbf{r}_i and \mathbf{q} by $\tau_i = d(\mathbf{r}_i, \mathbf{q})$.

Choice of Metric Post-Normalization. A similar metric, using ℓ_2 distance between normalized vectors (as opposed to ℓ_1), has been studied in prior work [20, 138] and even validated in a user study in [138]. However, as observed in [139], the ℓ_2 distance between distributions has the drawback that it could be small even for distributions with disjoint support. The ℓ_1 distance metric over discrete probability distributions has a direct correspondence with the traditional statistical distance metric known as *total variation distance* [140] and does not suffer from this drawback.

KL-divergence is another possibility as a distance metric, but it has the drawback that it will be infinite for any candidate that places 0 mass in a place where the target places nonzero mass, making it difficult to compare these (note that this follows directly from the definition: $KL(p\|q) = -\sum_i p_i \log \frac{q_i}{p_i}$).

5.2.2 Guarantees and Problem Statement

Since FASTMATCH takes samples to estimate the candidate histogram visualizations, and therefore may return incorrect results, we need to enforce probabilistic guarantees on the correctness of the returned results.

First, we introduce some notation: we use \mathbf{r}_i to denote the *estimate* of the candidate visualization, while \mathbf{r}_i^* (with normalized version $\bar{\mathbf{r}}_i^*$) is the *true* candidate visualization on the entire dataset. Our formulation also relies on constants ε , δ , and σ , which we assume either built into the system or provided by the analyst. We further use N and N_i to denote the total number of datapoints and number of datapoints corresponding to candidate i , respectively.

Guarantee 5.1. (*SEPARATION*) *Any approximate histogram \mathbf{r}_i with selectivity $\frac{N_i}{N} \geq \sigma$ that is in the true top- k closest (w.r.t. [Definition 5.2](#)) but not part of the output will be less than ε closer to the target than the furthest histogram that is part of the output. That is, if the algorithm outputs histograms $\mathbf{r}_{j_1}, \mathbf{r}_{j_2}, \dots, \mathbf{r}_{j_k}$, then, for all i , $\max_{1 \leq l \leq k} \{d(\mathbf{r}_{j_l}^*, \mathbf{q})\} - d(\mathbf{r}_i^*, \mathbf{q}) < \varepsilon$, or $\frac{N_i}{N} < \sigma$.*

Note that we use “selectivity” as a number and not as a property, matching typical usage in database systems literature [141, 142]. As such, candidates with lower selectivity appear less frequently in the data than candidates with higher selectivity.

Guarantee 5.2. (*RECONSTRUCTION*) *Each approximate histogram \mathbf{r}_i output as one of the top- k satisfies $d(\mathbf{r}_i, \mathbf{r}_i^*) < \varepsilon$.*

The first guarantee says that any ordering mistakes are relatively innocuous: for any two histograms \mathbf{r}_i and \mathbf{r}_j , if the algorithm outputs \mathbf{r}_j but not \mathbf{r}_i , when it should have been the other way around, then either $|d(\mathbf{r}_i^*, \mathbf{q}) - d(\mathbf{r}_j^*, \mathbf{q})| < \varepsilon$, or $\frac{N_i}{N} < \sigma$. The intuition behind the minimum selectivity parameter, σ , is that certain candidates may not appear frequently enough within the data to get a reliable reconstruction of the true underlying distribution responsible for generating the original data, and thus may not be suitable for downstream decision-making. For example, in our income example, a country with a population of 100 may have a histogram similar to the visual target but this would not be statistically significant. Overall, our guarantee states that we still return a visualization that is quite close to \mathbf{q} , and we can be confident that anything dramatically closer has relatively few total datapoints available within the data (i.e., N_i is small).

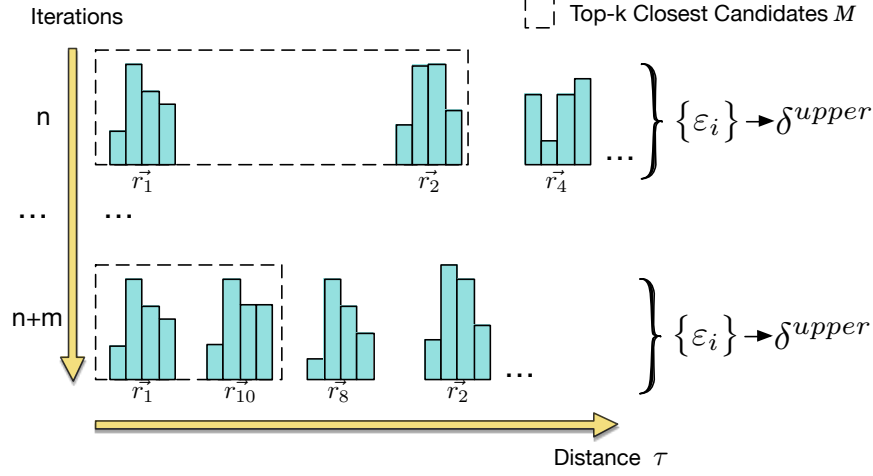


Figure 5.4: Illustration of HistSim.

The second guarantee says that the histograms output are not too dissimilar from the corresponding true distributions that would result from a complete scan of the data. As a result, they form an adequate and accurate proxy from which insights may be derived. With these definitions in place, we now formally state our core problem:

Problem 5.1. (*TOP-K-SIMILAR*). *Given a visual target \mathbf{q} , a histogram-generating query template, k , ε , δ , and σ , display k candidate attribute values $\{z_i\} \subseteq V_Z$ (and accompanying visualizations $\{\mathbf{r}_i\}$) as quickly as possible, such that the output satisfies [Guarantees 5.1 and 5.2](#) with probability greater than $1 - \delta$.*

5.3 THE HISTSIM ALGORITHM

In this section, we discuss how to conceptually solve [Problem 5.1](#). We outline an algorithm, named HistSim, which allows us to determine confidence levels for whether our separation and reconstruction guarantees hold. We rigorously prove in this section that when our algorithm terminates, it gives correct results with probability greater than $1 - \delta$ *regardless* of the data given as input. Many systems-level details and other heuristics used to make HistSim perform particularly well in practice will be presented in [Section 5.4](#). [Table 5.1](#) summarizes notation.

5.3.1 Algorithm Outline

HistSim operates by sampling tuples. Each of these tuples contributes to one or more candidate histograms, using which HistSim constructs histograms $\{\bar{\mathbf{r}}_i\}$. After taking enough samples

Algorithm 5.1: The HistSim algorithm

Input: Columns Z, X , visual target \mathbf{q} , parameters $k, \varepsilon, \delta, \sigma$

Output: Estimates M of the top- k closest candidates to \mathbf{q} , histograms $\{\mathbf{r}_i\}$

```
1
2 Initialization.
3  $n_i, n_i^\partial \leftarrow 0, \mathbf{r}_i, \mathbf{r}_i^\partial \leftarrow \mathbf{0}$  for  $1 \leq i \leq |V_Z|$ ;
4
5 stage 1:  $\delta^{upper} \leftarrow \frac{\delta}{3}$ ;
6 Repeat  $m$  times: uniformly randomly sample some tuple without replacement;
7 Update  $\{n_i\}, \{\mathbf{r}_i\}, \{\tau_i\}$  based on the new samples;
8  $\Delta \leftarrow \{\delta_i\}$  where  $\delta_i = \sum_{j=0}^{n_i} f(j; N, \lceil \sigma N \rceil, m)$  for  $1 \leq i \leq |V_Z|$ ;
9 Perform a Holm-Bonferroni statistical test with P-values in  $\Delta$ ; that is:
10  $A \leftarrow \{i : \delta_i \leq \frac{\delta}{|V_Z|-i+1} \text{ and for all } j < i, \delta_j \leq \frac{\delta}{|V_Z|-j+1}\}$ ;
11
12 stage 2:  $\delta^{upper} \leftarrow \frac{\delta}{3}$ ;
13 do
14    $\delta^{upper} \leftarrow \frac{1}{2}\delta^{upper}$ ;
15    $n_i += n_i^\partial, \mathbf{r}_i += \mathbf{r}_i^\partial, \tau_i \leftarrow d(\mathbf{r}_i, \mathbf{q})$  for  $i \in A$ ;
16    $n_i^\partial \leftarrow 0, \mathbf{r}_i^\partial \leftarrow \mathbf{0}$  for  $i \in A$ ;
17    $M \leftarrow \{i \in A : \tau_i \text{ among } k \text{ smallest}\}$ ;
18    $s \leftarrow \frac{1}{2}(\max_{i \in M} \tau_i + \min_{j \in A \setminus M} \tau_j)$ ;
19   Repeat: take uniform random samples from any  $i \in A$ ;
20   Update  $\{n_i^\partial\}, \{\mathbf{r}_i^\partial\}$ , and  $\{\tau_i^\partial\}$  based on the new samples;
21    $\varepsilon_i \leftarrow s + \frac{\varepsilon}{2} - \tau_i^\partial$  for  $i \in M$ ;
22    $\varepsilon_j \leftarrow \tau_j^\partial - (s - \frac{\varepsilon}{2})$  if  $s - \frac{\varepsilon}{2} \geq 0$  else  $\infty$  for  $j \in A \setminus M$ ;
23    $\Delta \leftarrow \{\delta_i\}$  where  $\delta_i \geq \mathbb{P}(d(\mathbf{r}_i^\partial, \mathbf{r}_i^*) > \varepsilon_i)$  for  $i \in A$ ;
24 while  $\max(\Delta) > \delta^{upper}$ ;
25
26 stage 3: Sample until  $n_i \geq \frac{2}{\varepsilon^2} (|V_X| \log 2 + \log \frac{3k}{\delta})$ , for all  $i \in M$ ;
27 Update  $\{\mathbf{r}_i\}$  based on the new samples;
28 return  $M$ , and  $\{\mathbf{r}_i : i \in M\}$ ;
```

corresponding to each candidate, it will eventually be likely that $d(\mathbf{r}_i, \mathbf{r}_i^*)$ is “small”, and that $|d(\mathbf{r}_i, \mathbf{q}) - d(\mathbf{r}_i^*, \mathbf{q})|$ is likewise “small”, for each i . More precisely, the set of candidates will likely be in a state such that [Guarantees 5.1](#) and [5.2](#) are both satisfied simultaneously.

Stages Overview. HistSim separates its sampling into three stages, each with an error probability of at most $\frac{\delta}{3}$, giving an overall error probability of at most δ :

- Stage 1 [Prune Rare Candidates]: Sample datapoints uniformly at random without replacement, so that each candidate is sampled a number of times roughly proportional to the number of

datapoints corresponding to that candidate. Identify rare candidates that likely satisfy $\frac{N_i}{N} < \sigma$, and prune these ones.

- Stage 2 [Identify Top- k]: Take samples from the remaining candidates until the top- k have been identified reliably.
- Stage 3 [Reconstruct Top- k]: Sample from the estimated top- k until they have been reconstructed reliably.

This separation is important for performance: the pruning step (stage 1) often *dramatically reduces* the number of candidates that need to be considered in stages 2 and 3.

The first two stages of HistSim factor into phases that are pure I/O and phases that involve one or more statistical tests. The I/O phases sample tuples (lines 6 and 19 in Algorithm 5.1)—we will describe how in Section 5.4; our algorithm’s correctness is independent of how this happens, provided that the samples are uniform.

Stage 1: Pruning Rare Candidates (Section 5.3.3). During stage 1, the I/O phase (line 6) takes m samples, for some m fixed ahead of time. This is followed by updating, for each candidate i , the number of samples n_i observed so far (line 7), and using the P-values $\{\delta_i\}$ of a test for underrepresentation to determine whether each candidate i is rare, i.e., has $\frac{N_i}{N} < \sigma$ (lines 7 to 9).

Stage 2: Identifying Top- k (Section 5.3.4). For stage 2, we focus on a smaller set of candidates; namely, those that we did not find to be rare (denoted by A). Stage 2 is divided into *rounds*. Each round attempts to use existing samples to estimate which candidates are top- k and which are non top- k , and then draws new samples, testing how unlikely it is to observe the new samples in the event that its guess of the top- k is wrong. If this event is unlikely enough, then it has recovered the correct top- k , with high probability.

At the start of each round, HistSim accumulates any samples taken during the previous round (lines 15 to 16). It then determines the current top- k candidates and a separation point s between top- k and non top- k (lines 17 to 18), as this separation point determines a set of hypotheses to test. Then, it begins an I/O phase and takes samples (line 19). The samples taken each round are used to generate the number of samples taken per candidate, $\{n_i^\partial\}$, the estimates $\{\mathbf{r}_i^\partial\}$, and the distance estimates $\{\tau_i^\partial\}$ (line 20). These statistics are computed from fresh samples each round (i.e., they do not reuse samples across rounds) so that they may be used in a statistical test (lines 20 to 23), discussed in Section 5.3.4. After computing the P-values for each null hypothesis to test (line 23), HistSim determines whether it can reject all the hypotheses with type 1 error (i.e., probability of mistakenly rejecting a true null hypothesis) bounded by δ^{upper} and break from the loop (line 24). If not, it repeats with new samples and a smaller δ^{upper} (where the $\{\delta^{upper}\}$ are chosen so that the probability of error across *all* rounds is at most $\frac{\delta}{3}$).

Stage 3: Reconstructing Top- k (Section 5.3.5). Finally, stage 3 ensures that the identified top- k , M , all satisfy $d(\mathbf{r}_i, \mathbf{r}_i^*) \leq \varepsilon$ for $i \in M$ (so that **Guarantee 5.2** holds), with high probability.

Figure 5.4 illustrates HistSim stage 2 running on a toy example in which we compute the top-2 closest histograms to a target. At round n , it estimates \mathbf{r}_1 and \mathbf{r}_2 as the top-2 closest, which it refines by the time it reaches round $n + m$. As the rounds increase, HistSim takes more samples to get better estimates of the distances $\{\tau_i\}$ and thereby improve the chances of termination when it performs its multiple hypothesis test in stage 2.

Choosing where to sample and how many samples to take. The estimates M and $\{\tau_i\}$ allow us to determine which candidates are “important” to sample from in order to allow termination with fewer samples; we return to this in **Section 5.4**. Our HistSim algorithm is agnostic to the sampling approach.

Outline. We first discuss the Holm-Bonferroni method for testing multiple statistical hypotheses simultaneously in **Section 5.3.2**, since stage 1 of HistSim uses it as a subroutine, and since the simultaneous test in stage 2 is based on similar ideas. In **Section 5.3.3**, we discuss stage 1 of HistSim, and prove that upon termination, all candidates i flagged for pruning satisfy $\frac{N_i}{N} < \sigma$ with probability greater than $\frac{\delta}{3}$. Next, in **Section 5.3.4**, we discuss stage 2 of HistSim, and prove that upon termination, we have the guarantee that any non-pruned candidate mistakenly classified as top- k is no more than ε further from the target than the furthest true non-pruned top- k candidate (with high probability). The proof of correctness for stage 2 is the most involved and is divided as follows:

- In **Section 5.3.4**, we give lemmas that allow us to relate the reconstruction of the candidate histograms from estimates $\{\mathbf{r}_i^\partial\}$ to the separation guarantee via multiple hypothesis testing;
- In **Section 5.3.4**, we describe a method to select appropriate hypotheses to use for testing in the lemmas of **Section 5.3.4**;
- In **Section 5.3.4**, we prove a theorem that enables us to use the samples per candidate histogram to determine the P-values associated with the hypotheses.

In **Section 5.3.5**, we discuss stage 3 and conclude with an overall proof of correctness.

5.3.2 Controlling Family-wise Error

In the first two stages of HistSim, the algorithm needs to perform multiple statistical tests simultaneously [143]. In stage 1, HistSim tests null hypotheses of the form “candidate i is high-selectivity” versus alternatives like “candidate i is not high-selectivity”. In this case, “rejecting the null hypothesis at level δ^{upper} ” roughly means that the probability that candidate i is high-selectivity is at most δ^{upper} . Likewise, during stage 2, HistSim tests null hypotheses of the form “candidate i ’s true distance from \mathbf{q} , τ_i^* , lies above (or below) some fixed value s .” If the algorithm correctly

rejects every null hypothesis while controlling the family-wise error [144] at level δ^{upper} , then it has correctly determined which side of s every τ_i^* lies, a fact that we use to get the separation guarantee.

Since stages 1 and 2 test multiple hypotheses at the same time, HistSim needs to control the family-wise type 1 error (false positive) rate of these tests simultaneously. That is, if the family-wise type 1 error is controlled at level δ^{upper} , then the probability that one or more rejecting tests in the family should not have rejected is less than δ^{upper} — during stage 1, this intuitively means that the probability one or more high-selectivity candidates were deemed to be low-selectivity is at most δ^{upper} , and during stage 2, this roughly means that the probability of selecting some candidate as top- k when it is non top- k (or vice-versa) is at most δ^{upper} .

The reader may be familiar with the Bonferroni correction, which enforces a family-wise error rate of δ^{upper} by requiring a significance level $\frac{\delta^{upper}}{|V_Z|}$ for each test in a family with $|V_Z|$ tests in total. We instead use the Holm-Bonferroni method [145], which is uniformly more powerful than the Bonferroni correction, meaning that it needs fewer samples to make the same guarantee. Like its simpler counterpart, it is correct regardless of whether the family of tests has any underlying dependency structure. In brief, a level δ^{upper} test over a family of size $|V_Z|$ works by first sorting the P-values $\{\delta_i\}$ of the individual tests in increasing order, and then finding the minimal index j (starting from 1) where $\delta_j > \frac{\delta^{upper}}{|V_Z|-j+1}$ (if this does not exist, then set $j = |V_Z|$). The tests with smaller indices reject their respective null hypotheses at level δ^{upper} , and the remaining ones do not reject.

5.3.3 Stage 1: Pruning Rare Candidates

One way to remove *rare* (i.e. low-selectivity) candidates from processing is to use an index to look up how many tuples correspond to each candidate. While this will work well for some queries, it unfortunately does not work in general, as candidates generated from queries of the form in [Definition 5.1](#) could have arbitrary predicates attached, which cannot all be indexed ahead-of-time. Thus, we turn to sampling.

To prune rare candidates, we need some way to determine whether each candidate i satisfies $\frac{N_i}{N} < \sigma$ with high probability. To do so, we make the simple observation that, after drawing m tuples without replacement uniformly at random, the number of tuples corresponding to candidate i follows a hypergeometric distribution [146]. The number of samples to take, m , is a parameter; we observe in our experiments that $m = 5 \cdot 10^5$ is an appropriate choice.¹ That is, if candidate i has N_i total corresponding tuples in a dataset of size N , then the number of tuples n_i for candidate i in a uniform sample without replacement of size m is distributed according to $n_i \sim \text{HypGeo}(N, N_i, m)$.

¹Our results are not sensitive to the choice of m , provided m is not too small (so that the algorithm fails to prune anything) or too big (i.e., a nontrivial fraction of the data).

As such, we can make use of a well-known test for underrepresentation [144] to accurately detect when candidate i has $\frac{N_i}{N} < \sigma$. The null hypothesis is that candidate i is not underrepresented (i.e., has $N_i \geq \sigma N$), and letting $f(\cdot; N, \lceil \sigma N \rceil, m)$ denote the hypergeometric pdf in this case, the P-value for the test is given by

$$\sum_{j=0}^{n_i} f(j; N, \lceil \sigma N \rceil, m) \quad (5.3)$$

where n_i is the number of observed tuples for candidate i in the sample of size m . Roughly speaking, the P-value measures how surprised we are to observe n_i or fewer tuples for candidate i when $\frac{N_i}{N} \geq \sigma$ — the lower the P-value, the more surprised we are.

If we reject the null hypothesis for some candidate i when the P-value is at most δ_i , we are claiming that candidate i satisfies $\frac{N_i}{N} < \sigma$, and the probability that we are wrong is then at most δ_i . Of course, we need to test *every* candidate for rareness, not just a given candidate, which is why HistSim stage 1 uses a Holm-Bonferroni procedure to control the *family-wise* error at any given threshold. We note in passing that the joint probability of observing n_i samples for candidate i across *all* candidates is a multivariate hypergeometric distribution for which we could perform a similar test without a Holm-Bonferroni procedure, but the CDF of a multivariate hypergeometric is extremely expensive to compute, and we can afford to sacrifice some statistical power for the sake of computational efficiency since we only need to ensure that the candidates pruned are actually rare, without necessarily finding all the rare candidates — that is, we need high precision, not high recall.

We now prove a lemma regarding correctness of stage 1.

Lemma 5.1 (Stage 1 Correctness). *After HistSim stage 1 completes, every candidate i removed from A satisfies $\frac{N_i}{N} < \sigma$, with probability greater than $1 - \frac{\delta}{3}$*

Proof. This follows immediately from the above discussion, in conjunction with the fact that the P-values generated from each test for underrepresentation are fed into a Holm-Bonferroni procedure that operates at level $\frac{\delta}{3}$, so that the probability of pruning one or more non-rare candidates is bounded above by $\frac{\delta}{3}$.

5.3.4 Stage 2: Identifying Top- K Candidates

HistSim stage 2 attempts to find the top- k closest to the target out of those remaining after stage 1. To facilitate discussion, we first introduce some definitions.

Definition 5.3. (*MATCHING CANDIDATES*) *A candidate is called matching if its distance estimate $\tau_i = d(\mathbf{r}_i, \mathbf{q})$ is among the k smallest out of all candidates remaining after stage 1.*

We denote the (dynamically changing) set of candidates that are matching during a run of HistSim as M ; we likewise denote the true set of matching candidates out of the remaining, non-pruned candidates in A as M^* . Next, we introduce the notion of ε_i -deviation.

Definition 5.4. (ε_i -DEVIATION) *The empirical vector of counts \mathbf{r}_i for some candidate i has ε_i -deviation if the corresponding normalized vector $\bar{\mathbf{r}}_i$ is within ε_i of the exact distribution $\bar{\mathbf{r}}_i^*$. That is, $d(\mathbf{r}_i, \mathbf{r}_i^*) = \|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 < \varepsilon_i$*

Note that **Definition 5.4** overloads the symbol ε to be candidate-specific by appending a subscript. In **Section 5.3.4**, we provide a way to quantify ε_i given samples.

If HistSim reaches a state where, for each matching candidate $i \in M$, candidate i has ε_i -deviation, and $\varepsilon_i < \varepsilon$ for all $i \in M$, then it is easy to see that the **Guarantee 5.2** holds for the matching candidates. That is, in such a state, if HistSim output the histograms corresponding to the matching candidates, they would look similar to the true histograms. In the following sections, we show that ε_i -deviation can also be used to achieve **Guarantee 5.1**.

Notation for Round-Specific Quantities. In the following subsections, we use the superscript “ Δ ” to indicate quantities corresponding to samples taken during a particular round of HistSim stage 2, such as $\{\mathbf{r}_i^\Delta\}$ and $\{\tau_i^\Delta\}$. In particular, these quantities are *completely independent* of samples taken during previous rounds.

Deviation-Bounds Imply Separation

In order to reason about the separation guarantee, we prove a series of lemmas following the structure of reasoning given below:

- We show that when a carefully chosen set of null hypotheses are all false, M contains valid top- k closest candidates.
- Next, we show how to use ε_i -deviation to upper bound the probability of rejecting a *single* true null hypothesis.
- Finally, we show how to reject *all* null hypotheses while controlling the probability of rejecting *any* true ones.

Lemma 5.2 (False Nulls Imply Separation). *Consider the set of null hypotheses $\{H_0^{(i)}\}$ defined as follows, where $s \in \mathbb{R}^+$:*

$$H_0^{(i)} = \begin{cases} \tau_i^* \geq s + \frac{\varepsilon}{2}, & \text{for } i \in M \\ \tau_i^* \leq s - \frac{\varepsilon}{2}, & \text{for } i \in A \setminus M \end{cases} \quad (5.4)$$

*When $H_0^{(i)}$ is false for every $i \in A$, then M is a set of top- k candidates that is correct with respect to **Guarantee 5.1**.*

Proof. When all the null hypotheses are false, then $\tau_i^* < s + \frac{\varepsilon}{2}$ for all $i \in M$, and $\tau_j^* > s - \frac{\varepsilon}{2}$ for all $j \in A \setminus M$. This means that

$$\max_{i \in M} \tau_i^* - \min_{j \in A \setminus M} \tau_j^* < \varepsilon \quad (5.5)$$

and thus M is correct with respect to the separation guarantee.

Intuitively, [Lemma 5.2](#) states that when there is some reference point s such that all of the candidates in M have their τ_i^* smaller than $s - \frac{\varepsilon}{2}$, and the rest have their τ_i^* greater than $s + \frac{\varepsilon}{2}$, then we have our separation guarantee.

Next, we show how to compute P-values for a single null hypothesis of the type given in [Lemma 5.2](#). Below, we use “ \mathbb{P}_H ” to denote the probability of some event when hypothesis H is true.

Lemma 5.3 (Distance Deviation Testing). *Let $x \in \mathbb{R}^+$. To test the null hypothesis $H_0^{(i)} : \tau_i^* \geq x$ versus the alternative $H_A^{(i)} : \tau_i^* < x$, we have that, for any $\varepsilon_i > 0$,*

$$\mathbb{P}_{H_0^{(i)}} \left[x - \tau_i^\partial > \varepsilon_i \right] \leq \mathbb{P} \left(d(\mathbf{r}_i^\partial, \mathbf{r}_i^*) > \varepsilon_i \right) \quad (5.6)$$

Likewise, for testing $H_0^{(i)} : \tau_i^ \leq x$ versus the alternative $H_A^{(i)} : \tau_i^* > x$, we have*

$$\mathbb{P}_{H_0^{(i)}} \left[\tau_i^\partial - x > \varepsilon_i \right] \leq \mathbb{P} \left(d(\mathbf{r}_i^\partial, \mathbf{r}_i^*) > \varepsilon_i \right) \quad (5.7)$$

Proof. We prove the first case; the second is symmetric. Suppose candidate i satisfies $\tau_i^* \geq x$ for some $x \in \mathbb{R}^+$. Then, if we take n_i^∂ samples from which we construct the random quantities \mathbf{r}_i^∂ and τ_i^∂ , we have that

$$\mathbb{P}_{H_0^{(i)}} \left[x - \tau_i^\partial > \varepsilon_i \right] \leq \mathbb{P} \left(\tau_i^* - \tau_i^\partial > \varepsilon_i \right) \quad (5.8)$$

$$= \mathbb{P} \left(\|\bar{\mathbf{r}}_i^* - \bar{\mathbf{q}}\| - \|\bar{\mathbf{q}} - \bar{\mathbf{r}}_i^\partial\| > \varepsilon_i \right) \quad (5.9)$$

$$\leq \mathbb{P} \left(\|\bar{\mathbf{r}}_i^* - \bar{\mathbf{r}}_i^\partial\| > \varepsilon_i \right) \quad (5.10)$$

$$= \mathbb{P} \left(d(\mathbf{r}_i^*, \mathbf{r}_i^\partial) > \varepsilon_i \right) \quad (5.11)$$

Each step follows from the fact that increasing the quantity to the left of the “ $>$ ” sign within the probability expression can only increase the probability of the event inside. The first step follows from the assumption that $\tau_i^* \geq x$, and the third step follows from the triangle inequality.

We use [Lemma 5.3](#) in conjunction with [Lemma 5.2](#) by using $s \pm \frac{\varepsilon}{2}$ for the reference x of [Lemma 5.3](#), for a particular choice of s (discussed in [Section 5.3.4](#)). For example, [Lemma 5.3](#) shows

that when we are testing the null hypothesis for $i \in M$ that $\tau_i^* \geq s + \frac{\varepsilon}{2}$ and we observe τ_i^∂ such that $0 < \varepsilon_i = s + \frac{\varepsilon}{2} - \tau_i^\partial$, we can use (any upper bound of) $\mathbb{P}(d(\mathbf{r}_i^*, \mathbf{r}_i^\partial) > \varepsilon_i)$ as a P-value for this test. That is, consider a tester with the following behavior, illustrated pictorially:

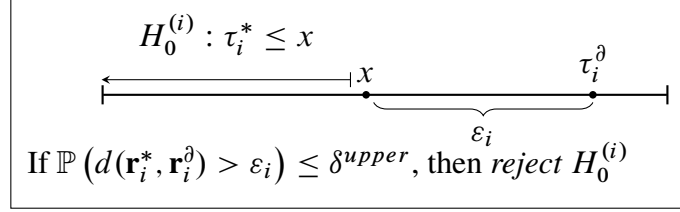


Figure 5.5: Depiction of hypothesis testing procedure which rejects for empirical histograms $\bar{\mathbf{r}}_i^\partial$ on the wrong side of and with distance τ_i^∂ sufficiently far from a particular x , informed by the split point s .

In the above figure, the tester assumes that τ_i^* is smaller than x , but it observes a value τ_i^∂ that exceeds x by ε_i . When the true value $\tau_i^* \leq x$ for any reference x , then the observed statistic τ_i^∂ will only be ε_i or larger than x (and vice-versa) when the reconstruction \mathbf{r}_i^∂ is also bad, in the sense that $\mathbb{P}(d(\mathbf{r}_i^*, \mathbf{r}_i^\partial) > \varepsilon_i)$ is very small. If the above tester rejects $H_0^{(i)}$ when $\mathbb{P}(d(\mathbf{r}_i^*, \mathbf{r}_i^\partial) > \varepsilon_i) \leq \delta^{upper}$, then [Lemma 5.3](#) says that it is guaranteed to reject a true null hypothesis with probability at most δ^{upper} . We discuss how to compute an upper bound on $\mathbb{P}(d(\mathbf{r}_i^*, \mathbf{r}_i^\partial) > \varepsilon_i)$ in [Section 5.3.4](#).

Finally, notice that [Lemma 5.3](#) provides a test which controls the type 1 error of an individual $H_0^{(i)}$, but we only know that the separation guarantee holds for $i \in M$ when *all* the hypotheses $\{H_0^{(i)}\}$ are false. Thus, the algorithm requires a way to control the type 1 error of a procedure that decides whether to reject every $H_0^{(i)}$ simultaneously. In the next lemma, we give such a tester which controls the error for any upper bound δ^{upper} .

Lemma 5.4 (Simultaneous Rejection). *Consider any set of null hypotheses $\{H_0^{(i)}\}$, and consider a set of P-values $\{\delta_i\}$ associated with these hypotheses. The tester given by*

$$\text{Decision} = \begin{cases} \text{reject every } H_0^{(i)}, & \text{when } \max_i \delta_i \leq \delta^{upper} \\ \text{reject no } H_0^{(i)}, & \text{otherwise} \end{cases} \quad (5.12)$$

rejects ≥ 1 true null hypotheses with probability $\leq \delta^{upper}$.

Proof. Consider the set of true null hypotheses and call it $\{H_0^{(t)}\}$ — suppose there are $T \geq 1$ in total (if $T = 0$, we have nothing to prove), and index them using t from 1 to T . Then

$$\mathbb{P}(\exists t : \text{reject } H_0^{(t)}) = \mathbb{P}(\forall t : \text{reject } H_0^{(t)}) \quad (5.13)$$

$$= \prod_{t=1}^T \mathbb{P}(\text{reject } H_0^{(t)} \mid \text{reject } H_0^{(1, \dots, t-1)}) \quad (5.14)$$

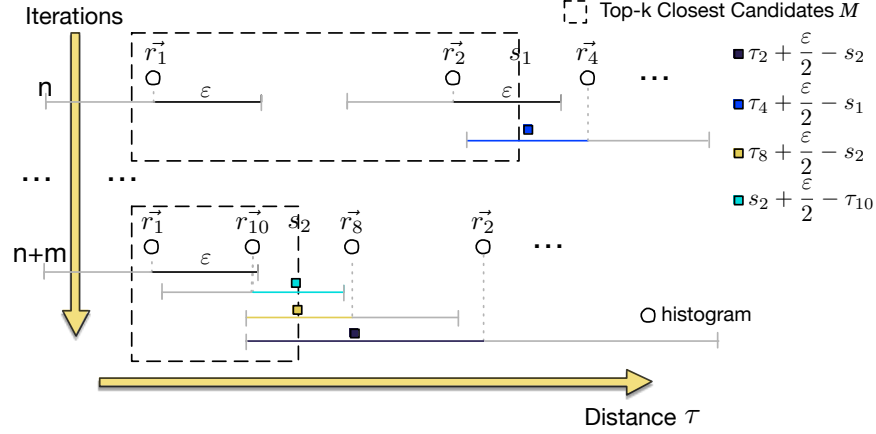


Figure 5.6: Illustration of HistSim choosing the split point s when testing whether the separation and reconstruction guarantees hold.

$$= \delta_1 \prod_{t=2}^T \mathbb{P} \left(\text{reject } H_0^{(t)} \mid \text{reject } H_0^{(1, \dots, t-1)} \right) \quad (5.15)$$

$$\leq \delta_1 \cdot 1 \quad (5.16)$$

$$\leq \delta^{upper} \quad (5.17)$$

The first step follows since null hypotheses are only rejected when they are all rejected. The second to last step follows since probabilities are at most 1, and the last step follows since the tester only rejects when all the P-values are at most δ^{upper} , including δ_1 .

Discussion of Lemma 5.4. At first glance, the multiple hypothesis tester given in Lemma 5.4, which compares all P-values to the same δ^{upper} , seems to be even more powerful than a Holm-Bonferroni tester, which compares P-values to various fractions of δ^{upper} . In fact, although based on similar ideas, they are not comparable: a Holm-Bonferroni tester may allow for rejection of a subset of the null hypotheses, whereas the tester of Lemma 5.4 is “all or nothing”. In fact, the tester of Lemma 5.4 is essentially the union-intersection method formulated in terms of P-values; see [143] for details.

Selecting Each Round’s Tests

Each round of HistSim stage 2 constructs a family of tests to perform whose family-wise error probability is at most δ^{upper} . At round t (starting from $t = 1$), δ^{upper} is chosen to be $\frac{\delta/3}{2^t}$, so that the error probability across *all* rounds is at most $\sum_{t \geq 1} \frac{\delta/3}{2^t} = \frac{\delta}{3}$ via a union bound (see Lemma 5.5 for details).

There is still one degree of freedom: namely, how to choose the split point s used for the null hypotheses in [Lemma 5.2](#). In [line 18](#), it is chosen to be $s \leftarrow \frac{1}{2}(\max_{i \in M} \tau_i + \min_{j \in A \setminus M} \tau_j)$. The intuition for this choice is as follows. Although the quantities \mathbf{r}_i^∂ and τ_i^∂ are generated from fresh samples in each round of HistSim stage 2, the quantities \mathbf{r}_i and τ_i are generated from samples taken across *all* rounds of HistSim stage 2. As such, as rounds progress (i.e., if the testing procedure fails to simultaneously reject multiple times), the estimates \mathbf{r}_i and τ_i become closer to \mathbf{r}_i^* and τ_i^* , the set M becomes more likely to coincide with M^* , and the null hypotheses $\{H_0^{(i)}\}$ chosen become less likely to be true *provided* an s chosen somewhere in $[\max_{i \in M} \tau_i, \min_{j \in A \setminus M} \tau_j]$, since values in this interval are likely to correctly separate M^* and $A \setminus M^*$ as more and more samples are taken. In the interest of simplicity, we simply choose the midpoint halfway between the furthest candidate in M and the closest candidate in $A \setminus M$. For example, at iteration n in [Figure 5.6](#), s lies halfway between candidates \mathbf{r}_2 and \mathbf{r}_4 . In practice, we observe that $\max_{i \in M} \tau_i$ and $\min_{j \in A \setminus M} \tau_j$ are typically very close to each other, so that the algorithm is not very sensitive to the choice of s , so long as it falls between M and $A \setminus M$.

[Figure 5.6](#) illustrates this choice of s and the $\{H_0^{(i)}\}$ on our toy example. As in [Figure 5.4](#), the boundary of M is represented by the dashed box. The split point s is located at the rightmost boundary of the dashed box. The $\{\varepsilon_j\}$ (i.e., the amounts by which the $\{\tau_j^\partial\}$ deviate from $s \pm \frac{\varepsilon}{2}$) determine the P-values associated with the $\{H_0^{(i)}\}$ which ultimately determine whether HistSim stage 2 can terminate, as we discuss more in the next section.

Deviation-Bounds Given Samples

The previous section provides us a way to check whether the rankings induced by the empirical distances $\{\tau_i\}$ are correct with high probability. This was facilitated via a test which measures our “surprise” for measuring $\{\tau_i^\partial\}$ if the current estimate M is not correct with respect to [Guarantee 5.1](#), which in turn used a test for how likely some candidate’s $d(\mathbf{r}_i^*, \mathbf{r}_i^\partial)$ is greater than some threshold ε_i after taking n_i samples. We now provide a theorem that allows us to infer, given the samples taken for a given candidate, how to relate ε_i with the probability δ_i with which the candidate can fail to respect its deviation-bound ε_i . The bound seems to be known to the theoretical computer science community as a “folklore fact” [[147](#)]; we give a proof for the sake of completeness. Our proof relies on repeated application of the method of bounded differences [[148](#)] in order to exploit some special structure in the ℓ_1 distance metric. The bound developed is *information-theoretically optimal*; that is, it takes asymptotically the fewest samples required to guarantee that an empirical distribution estimated from the samples will be no further than ε_i from the true distribution.

Theorem 5.1. Suppose we have taken n_i samples with replacement for some candidate i 's histogram, resulting in the empirical estimate \mathbf{r}_i . Then \mathbf{r}_i has ε_i -deviation with probability greater than $1 - \delta_i$ for $\varepsilon_i = \sqrt{\frac{2}{n_i} \left(|V_X| \log 2 + \log \frac{1}{\delta_i} \right)}$. That is, with probability $> 1 - \delta_i$, we have: $\|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 < \varepsilon_i$.

In fact, this theorem also holds if we sample without replacement; we return to this point in [Section 5.4](#). Please see the appendix ([§A.4](#)) for the proof.

Optimality of the bound in Theorem 5.1. If we solve for n_i in [Theorem 5.1](#), we see that we must have $n_i = \frac{|V_X| \log 4 + 2 \log(1/\delta_i)}{\varepsilon_i^2}$. That is, $\Omega\left(\frac{|V_X|}{\varepsilon_i^2}\right)$ samples are necessary guarantee that the empirical discrete distribution $\bar{\mathbf{r}}_i$ is no further than ε_i from the true discrete distribution $\bar{\mathbf{r}}_i^*$, with high probability. This matches the information theoretical lower bound noted in prior work [[139](#), [149](#), [150](#), [151](#)].

Generating P-values from Theorem 5.1. We use the above bound to generate P-values for testing the null hypotheses in [Lemma 5.2](#). From the discussion in that lemma, a tester which rejects $H_0^{(i)}$ for $i \in M$ when it observes $s + \frac{\varepsilon}{2} - \tau_i^\partial > \varepsilon_i$, for fixed ε_i , has a type 1 error bounded above by $\delta_i = 2^{|V_X|} \exp(-\varepsilon_i^2 n_i / 2)$. Since we want to bound the type 1 error rate by an amount δ^{upper} , this induces a particular ε_i against which we can compare $s + \frac{\varepsilon}{2} - \tau_i^\partial$, but because δ_i and ε_i are monotonically related, we can take

$$\delta_i = 2^{|V_X|} \exp\left(-(s + \frac{\varepsilon}{2} - \tau_i^\partial)^2 / 2\right) \quad (5.18)$$

and compare with δ^{upper} directly, allowing us to use this δ_i as a P-value for use with the tester in [Lemma 5.4](#).

Stage 2 Correctness

We can now show correctness of HistSim stage 2.

Lemma 5.5 (Stage 2 Correctness). After HistSim stage 2 completes, each candidate $i \in M$, satisfies $\tau_i^* - \tau_j^* \leq \varepsilon$ for every $j \in A \setminus M$ with probability greater than $1 - \frac{\delta}{3}$.

Proof. First, show that if HistSim stage 2 terminates after iteration t , then the probability of an error is at most $\frac{\delta/3}{2^t}$. Next, show that the probability of an error after terminating at *any* iteration is at most $\frac{\delta}{3}$ by union bounding over iterations.

If stage 2 terminates at iteration t , then the probability of rejecting one or more null hypotheses is at most $\frac{\delta/3}{2^t}$ by [Lemma 5.4](#) and by [Theorem 5.1](#). Each $H_0^{(i)}$ for $i \in M$ says that $\tau_i^* > s + \frac{\varepsilon}{2}$, and each $H_0^{(j)}$ for $j \in A \setminus M$ says that $\tau_j^* < s - \frac{\varepsilon}{2}$ – if all of these are false, then by [Lemma 5.2](#) we have that M and $A \setminus M$ induce a separation of the candidates that is correct with respect to [Guarantee 5.1](#), so the only way an error *could* occur is if one or more nulls are true. We just established that the

probability of rejecting one or more true nulls at iteration t is at most $\frac{\delta/3}{2^t}$, which means that the probability of an incorrect separation between M and $A \setminus M$ is also at most $\frac{\delta/3}{2^t}$.

Finally, by union bounding over iterations, we have that

$$\mathbb{P}(\cup_{t \geq 1} \text{mistake at iteration } t) \leq \sum_{t \geq 1} \mathbb{P}(\text{mistake at iteration } t) \quad (5.19)$$

$$< \sum_{t \geq 1} \frac{\delta/3}{2^t} = \delta/3 \quad (5.20)$$

Thus, when stage 2 terminates, M is correct (with respect to [Guarantee 5.1](#)) with probability greater than $1 - \frac{\delta}{3}$

5.3.5 Stage 3 and Overall Proof of Correctness

Stage 3 of HistSim, discussed in our overall proof of correctness, consists of taking samples from each candidate in M to ensure they all have ε -deviation with high probability (using [Theorem 5.1](#)). This proof is given next, and proceeds in four steps:

- Step 1: HistSim stage 1 incorrectly prunes one or more candidates meeting the selectivity threshold σ with probability at most $\frac{\delta}{3}$ ([Lemma 5.1](#)).
- Step 2: The probability that stage 2 incorrectly (with respect to [Guarantee 5.1](#)) separates M and $A \setminus M$ is at most $\frac{\delta}{3}$.
- Step 3: The probability that the set of candidates M violates [Guarantee 5.2](#) after stage 3 runs is at most $\frac{\delta}{3}$.
- Step 4: The union bound over any of these bad events occurring gives an overall error probability of at most δ .

Theorem 5.2. *The k histograms returned by [Algorithm 5.1](#) satisfy [Guarantees 5.1](#) and [5.2](#) with probability greater than $1 - \delta$.*

Proof. From [Lemma 5.1](#), the probability that high-selectivity candidates were pruned during stage 1 is upper bounded by $\frac{\delta}{3}$. From [Lemma 5.5](#), the probability that the algorithm chooses M such that there exists some $i \in M$ and $j \in M^* \setminus M$ with $\tau_i^* - \tau_j^* > \varepsilon$ is at most $\frac{\delta}{3}$. Union bounding over these events, the probability of either occurring is at most $\frac{2\delta}{3}$. Since [Guarantee 5.1](#) cannot be violated when neither of these events occur, the algorithm violates this guarantee also with probability at most $\frac{2\delta}{3}$. Finally, using [Theorem 5.1](#), HistSim stage 3 [line 26](#) takes a number of samples for each candidate $i \in M$ such that the probability that a given candidate fails to be reconstructed with error ε or less (that is, $d(\mathbf{r}_i, \mathbf{r}_i^*) > \varepsilon$) is at most $\frac{\delta}{3k}$. Union bounding over all candidates in M , and

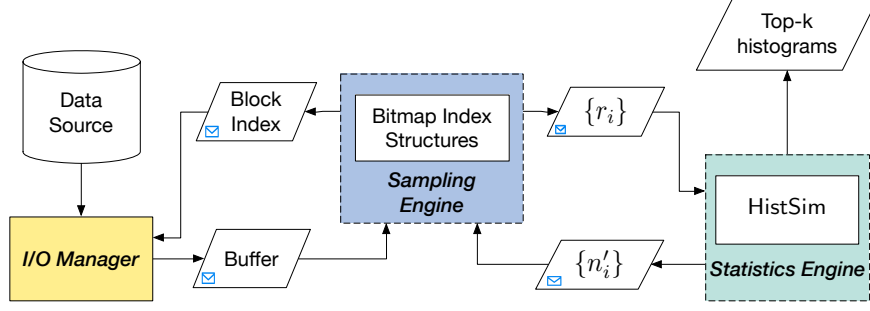


Figure 5.7: FASTMATCH system architecture

noting that $|M| = k$, the probability that one or more candidates does not have ε_i -deviation is at most $\frac{\delta}{3}$. Union bounding with the upper bound on the probability that [Guarantee 5.1](#) is violated, the probability that either [Guarantee 5.1](#) or [Guarantee 5.2](#) is violated is at most $\frac{2\delta}{3} + \frac{\delta}{3} = \delta$, and we are done.

Computational Complexity. Stage 1 of [Algorithm 5.1](#) shares computation between candidates when computing P-values induced by the hypergeometric distribution, and thus makes at most $\max_{i \in V_Z} n_i$ calls to evaluate a hypergeometric pdf (we use Boost’s implementation [[152](#)]); this can be done in $\mathcal{O}(\max_{i \in V_Z} n_i)$. To facilitate the sharing, stage 1 requires sorting the candidates in increasing order of n_i , which is $\mathcal{O}(|V_Z| \cdot \log |V_Z|)$. Next, each iteration of HistSim stage 2 requires computing distance estimates τ_i and τ_i^δ for every $i \in A$, which runs in time $\mathcal{O}(|A| \cdot |V_X|)$. Each iteration of stage 2 further uses a sort of candidates in A by τ_i to determine M and s , which is $\mathcal{O}(|A| \cdot \log |A|)$. HistSim stage 2 almost always terminates within 4 or 5 iterations in practice. Overall, we observe that the computation required is inexpensive compared to the cost of I/O, even for data stored in-memory.

5.4 THE FASTMATCH SYSTEM

This section describes FASTMATCH, which implements the HistSim algorithm. We start by presenting the high-level components of FASTMATCH. We then describe the challenges we faced while implementing FASTMATCH and describe how the components interact to alleviate those challenges, while still satisfying [Guarantees 5.1](#) and [5.2](#). While design choices presented in this section are heuristics with practicality in mind, the algorithm implemented is still theoretically rigorous, with results satisfying our probabilistic guarantees. In the following, each time we describe a heuristic, we will clearly point it out as such.

5.4.1 FASTMATCH Components

FASTMATCH has three key components: the I/O Manager, the Sampling Engine, and the Statistics engine. We describe each of them in turn; [Figure 5.7](#) provides an architecture diagram—we will revisit the interactions within the diagram at the end of the section.

I/O Manager. In FASTMATCH, requests for I/O are serviced at the granularity of *blocks*. The I/O manager simply services requests for blocks in a synchronous fashion. Given the location of some block, it synchronously processes the block at that location.

Sampling Engine. The sampling engine is responsible for deciding which blocks to sample. It uses bitmap index structures (described below) in order to determine the types of samples located at a given block. Given the current state of the system, it prioritizes certain candidates over others for sampling.

Statistics Engine. The statistics engine implements most of the logic in the HistSim algorithm. The only substantial difference between the actual code and the pseudocode presented in [Algorithm 5.1](#) is that the statistics engine does not actually perform any sampling, instead leaving this responsibility to the sampling engine. The reason for separating these components will be made clear later on.

Bitmap Index Structures. FASTMATCH runs on top of a bitmap-based sampling system used for sampling on-demand, as in prior work [[5](#), [6](#), [11](#), [24](#)]. These papers have demonstrated that bitmap indexes [[153](#)] are effective in supporting sampling for incremental or early termination of visualization generation. Within FASTMATCH, bitmap indexes help the sampling engine determine whether a given block contains samples for a given candidate. For each attribute A , and each attribute value A_v , we store a bitmap, where a ‘0’ at position p indicates that the corresponding block at position p contains no tuples with attribute value A_v , and a ‘1’ indicates that block p contains one or more tuples with attribute value A_v . Candidate visualizations are generated by attribute values (or a predicate of ANDs and ORs over attribute values; see [Appendix C](#)), so these bitmaps allow the sampling engine to rapidly test whether a block contains tuples for a given candidate histogram. Bitmaps are amenable to significant compression [[61](#), [154](#)], and since we are further only requiring a single bit per block per attribute value, our storage requirements are orders-of-magnitude cheaper than past work that requires a bit per tuple [[5](#), [6](#), [24](#)]. Notice also that our techniques also apply for continuous candidate attributes; please see [Appendix C](#) for details.

5.4.2 Implementation Challenges

So far, we have designed HistSim without worrying about how sampling actually takes place, with an implicit assumption that there is no overhead to taking samples randomly across various

candidates. While implementing HistSim within FASTMATCH, we faced several non-trivial challenges, outlined below:

- **Challenge 1: Random sampling at odds with performance characteristics of storage media.** The cost to fetch data is locality-dependent when dealing with real storage devices. Even if the data is stored in-memory, tuples (i.e., samples) that are spatially closer to a given tuple may be cheaper to fetch, since they may already be present in CPU cache.
- **Challenge 2: Deciding how many samples to take between rounds of HistSim.** The HistSim algorithm does not specify how many samples to taken in between rounds of stage 2; it is agnostic to this choice, with correctness unaffected. If the algorithm takes many samples, it may spend more time on I/O than is necessary to terminate with a guarantee. If the algorithm does not take enough samples, the statistical test on [line 24](#) will probably not reject across many rounds, decaying δ^{upper} and making it progressively more difficult to get enough samples to meet stage 2’s termination criterion.
- **Challenge 3: Non-uniform cost/benefit of different candidates.** Tuples for some candidates can be over-represented in the data and therefore take less time to sample compared to underrepresented candidates. At the same time, the benefit of sampling tuples corresponding to different candidate histograms is non-uniform: for example, those histograms which are “far” from the target distribution are less useful (in terms of getting HistSim to terminate quickly) than those for which HistSim chooses small values for ε_i .
- **Challenge 4: Assessing benefit to candidates depends on data seen so far.** The “best” choice of which tuples to sample for getting HistSim to terminate quickly can be most accurately estimated from *all* the data seen so far, including the most recent data. However, computing this estimate after processing every tuple and blocking I/O until the “best” decision can be made is prohibitively expensive.

We now describe our approaches to tackling these three challenges.

Challenge 1: Randomness via Data Layout

To maximize performance benefits from locality, we randomly permute the tuples of our dataset as a preprocessing step, and to “sample” we may then simply perform a linear scan of the shuffled data starting from any point. This matches the assumptions of stage 1 of HistSim, which requires samples to be taken without replacement. Although the theory we developed in [Section 5.3](#) for HistSim stage 2 was for sampling with-replacement, as noted in [\[155, 156\]](#), it still holds now that we are sampling without replacement, as concentration results developed for the with-replacement regime may be transferred automatically to the without-replacement regime. This approach of randomly permuting upfront is not new, and is adopted by other approximate query processing systems [\[157, 158, 159\]](#).

Challenge 2: Deciding Samples to Take Between Rounds

The HistSim algorithm leaves the number of samples to take during a given round of stage 2 lines 19 unspecified; its correctness is guaranteed regardless of how this choice is made. This choice offers a tradeoff: take too many samples, and the system will spend a lot of time unnecessarily on I/O; take too few, and the algorithm will never terminate, since the “difficulty” of the test increases with each round, as we set $\delta^{upper} \leftarrow \delta^{upper} / 2$.

To combat this challenge, we employ a simple heuristic. To estimate the number of samples we need to take for candidate i , we assume that $\tau_i = \tau_i^*$, so that we need to learn \mathbf{r}_i^∂ to within ε'_i of \mathbf{r}_i^* for a given round’s statistical test to successfully reject, where $\varepsilon'_i = s + \frac{\varepsilon}{2} - \tau_i$ for $i \in M$ and $\varepsilon'_i = \tau_i - (s - \frac{\varepsilon}{2})$ for $i \in A \setminus M$. (Recall that we use ε_i -deviation to upper bound the P-values.) For this setting of $\{\varepsilon'_i\}$, we thus choose to take samples for each candidate by solving for n_i in the bound of [Theorem 5.1](#). This yields

$$n'_i = 2 (|V_X| \log 2 - \log \delta^{upper}) / (\varepsilon'_i)^2 \quad (5.21)$$

Each round of stage 2 of our FASTMATCH implementation of HistSim thus continues to take samples until $n_i^\partial \geq n'_i$ for every candidate i . It then performs the multiple hypothesis test on [lines 20 to 23](#). If it rejects, the algorithm terminates and the system gives the output to the user; otherwise, it once again estimates each n'_i using [Equation \(5.21\)](#) (plugging in $\{\varepsilon'_i\}$ from updated $\{\tau_i\}$) and repeats.

Challenge 3: Block Choice Policies

Deciding which blocks to read during stage 1 of HistSim is simple since we are only trying to detect low-selectivity candidates — in this case we just scan each block sequentially. Deciding which blocks to read during stage 2 of HistSim is more difficult due to the non-uniform cost (i.e., time) and benefit of samples for each candidate histogram. If either cost or benefit were uniform across candidates, matters would be simplified significantly: if cost were uniform, we could simply read in the blocks with the most beneficial candidates; if benefit were uniform, we could simply read in the lowest cost blocks (for example, those closest spatially to the current read position). To address these concerns, we developed a simple policy which we found worked well in practice for getting HistSim to terminate quickly.

AnyActive block selection policy. Recall that the end of each iteration of stage 2 of HistSim estimates the number of samples $\{n'_i\}$ necessary from each candidate so that the next iteration is more likely to terminate. Note that if each candidate satisfied $n_i = n'_i$ at the time HistSim performed the test for termination and *before* it computed the $\{n'_i\}$, then HistSim would be in a state where it can safely terminate. Those candidates for whom $n_i < n'_i$ we dub *active candidates*, and we employ a

very simple block selection policy, dubbed the AnyActive block selection policy, which is to *only read blocks which contain at least one tuple corresponding to some active candidate*. The bitmap indexes employed by FASTMATCH allow it to rapidly test whether a block contains tuples for a given candidate visualization, and thus to rapidly apply the AnyActive block selection policy. Overall, our approach is as follows: we read blocks in sequence, and if blocks satisfy our AnyActive criterion, then we read all of the tuples in that block, else, we skip that block. We discuss how to make this approach performant below.

A naïve variant of this policy is presented in [Algorithm 5.2](#), for which we now describe improvements.

Challenge 4: Asynchronous Block Selection

Algorithm 5.2: Naïve AnyActive block processing

Input: unpruned candidate set A , block index i

Output: A value indicating whether to :myread or :skip block i

```

1 for each active mycand  $\in A$  do
    // cache inefficient index lookup
    // evicts bits from previous candidate's bitmap index
2   if mycand.myindex_lookup(i) then
3       return :myread;
4   end
5 end
6 return :skip;
```

Algorithm 5.3: AnyActive block selection with lookahead

Input: lookahead amount, start block, unpruned candidate set A

Output: An array mark indicating whether to :read or :skip blocks

// Initialization

```

1 mark[i]  $\leftarrow$  :skip for  $0 \leq i < \text{lookahead}$ ;
2 for each active cand  $\in A$  do
3     for  $0 \leq i < \text{lookahead}$  do
4         if mark[i] == :read then
5             continue;
6         else if cand.index_lookup(start + i) then
7             mark[i]  $\leftarrow$  :read;
8         end
9     end
10 end
11 return mark
```

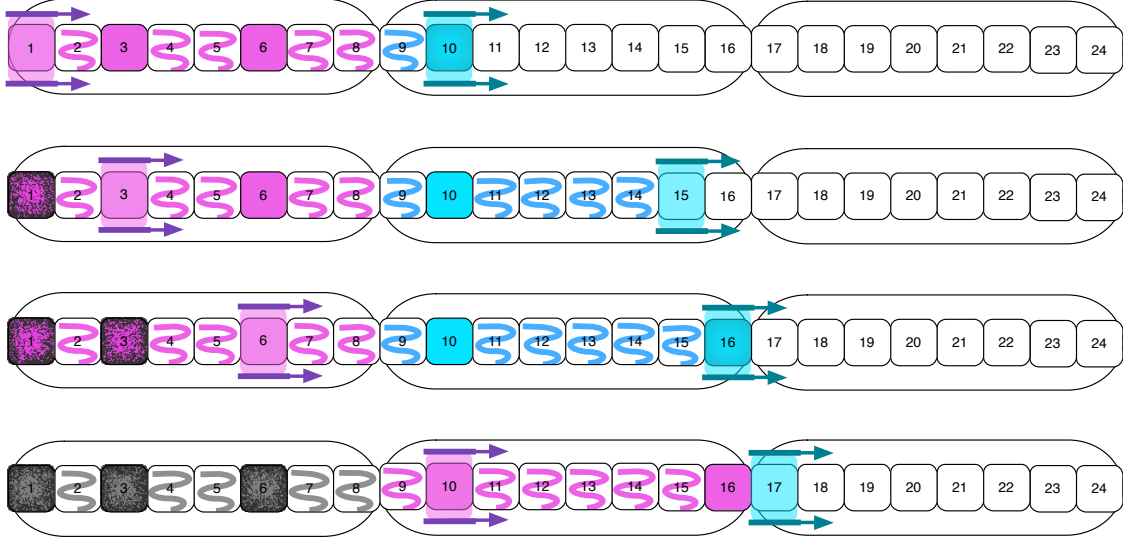


Figure 5.8: While the I/O manager processes magenta blocks, the sampling engine selects blue blocks ahead of time, using lookahead. Blocks with solid color = read, blocks with squiggles = skip.

From the previous discussion, the sampling engine employs an *AnyActive* block selection policy when deciding which blocks to process. Ideally, the $\{n_i\}$ and $\{n'_i\}$ (number of samples taken for candidate i and estimated number of samples needed for candidate i , respectively) used to assign active status to candidates should be computed from the freshest possible counts available to the sampling engine. That is, in an ideal setting, each candidate’s active status would be updated immediately after each block is read, and the potentially new active status should be used for making decisions about immediately subsequent blocks. Unfortunately, this requirement is at odds with real system characteristics. Employing it exactly implies leaving the I/O manager idle while the sampling engine determines whether each block should be read or skipped. To prevent this issue, we relax the requirement that the sampling thread employ *AnyActive* with the freshest $\{n_i\}$ available to it. Instead, given the current $\{n_i\}$ and fresh set of $\{n'_i\}$, it precomputes the active status for each candidate and “looks ahead”, marking an entire batch of blocks for either reading or skipping, and communicates this with the I/O manager. The batch size, or the *lookahead* amount, is a system parameter, and offers a trade-off between freshness of active states used for *AnyActive* and degree to which the I/O manager must idle while waiting for instructions on which block to read next. We evaluate the impact of this parameter in our experimental section. The lookahead process is depicted in [Figure 5.8](#) for a value of *lookahead* = 8. While the I/O manager processes a previously marked batch of magenta-colored lookahead blocks, the sampling engine’s lookahead thread marks the next batch in blue. It waits to mark the next batch until the I/O manager “catches up”.

Employing lookahead allows us to prevent two bottlenecks. First, the sampling engine need not wait for each candidate’s active status to update after a block is read before moving on to the next block, effectively decoupling it from the I/O manager.

The second bottleneck prevented by lookahead is more subtle. To illustrate it, consider the pseudocode in [Algorithm 5.2](#), implementing the AnyActive block policy. The AnyActive block policy algorithm works by considering each candidate in turn, and querying a bitmap index for that candidate to determine whether the current block contains tuples corresponding to that candidate. Querying a bitmap actually brings in surrounding bits into the cache of the CPU performing the query, and evicts whatever was previously in the cache line. If blocks are processed individually, then only a single bit in the bitmap is used each time a portion is brought into cache. This is quite wasteful and turns out to hurt performance significantly as we will see in the experiments. Instead, applying AnyActive selection to lookahead-size chunks instead of individual blocks is a better approach. This simply adds an extra inner loop to the procedure shown in [Algorithm 5.2](#) (depicted in [Algorithm 5.3](#)). This approach has much better cache performance, since it uses an entire cache-line’s worth of bits while employing AnyActive.

We verify in our experiments that these optimizations allow FASTMATCH to terminate more quickly via AnyActive block selection with *fresh-enough* active states without significantly slowing any single component of the system.

5.4.3 System Architecture

FASTMATCH is implemented within a few thousand lines of C++. It uses `pthread`s [160] for its threading implementation. FASTMATCH uses a column-oriented storage engine, as is common for analytics tasks. We can now complete our description of [Figure 5.7](#). When the I/O manager receives a request for a block at a particular block index from the sampling engine (via the “block index” message), it eventually returns a buffer containing the data at this block to the sampling engine (via the “buffer” message). Once the I/O phase of stage 1 or 2 of HistSim completes, the sampling engine sends the current per-group counts for each candidate, $\{\mathbf{r}_i\}$, to the statistics engine. After running a test for whether to move to stage 2 (performed in stage 1) or to terminate (performed in stage 2), the statistics engine either posts a message of updated n' (in stage 1) or $\{n'_i\}$ (stage 2) that the sampling engine uses to determine when to complete the I/O phase of each HistSim stage, as well as how to perform block selection during stage 2.

Dataset	Size	#Tuples	#Attributes	Replications
FLIGHTS	32 GiB	606 million	7	5×
TAXI	36 GiB	679 million	7	4×
POLICE	34 GiB	448 million	10	72×

Table 5.2: Descriptions of datasets used to evaluate FASTMATCH.

5.5 EMPIRICAL STUDY

The goal of our experimental evaluation is to test the accuracy and runtime of FASTMATCH against other approximate and exact approaches on a diverse set of real datasets and queries. Furthermore, we want to validate the design decisions that we made for FASTMATCH in Section 5.4 and evaluate their impact.

5.5.1 Datasets and Queries

We evaluate FASTMATCH on publicly available real-world datasets summarized in Table 5.2 — flight records [134], taxi trips [161], and police road stops [162]. The replication value indicates how many times each dataset was replicated to create a larger dataset. In preprocessing these datasets, we eliminated rows with “N/A” or erroneous values for any column appearing in one or more of our queries.

FLIGHTS Dataset. Our FLIGHTS dataset, representing delays measured for flights at more than 350 U.S. airports from 1987 up to 2008, is available at [134]; we used 7 attributes (for origin / destination airports, departure / arrival delays, day of week, day of month, and departure hour).

TAXI Dataset. Our TAXI dataset summarizes all Yellow Cab trips in New York in 2013 [161]. The subset of data we used corresponds with the urls ending in “yellow_tripdata_2013” in the file `raw_data_urls.txt`. We extracted some time-based discrete attributes, two attributes based on passenger count, and one attribute based on area, for 7 columns total. In particular, the “Location” attribute was generated by binning the pickup location into regions of 0.01 longitude by 0.01 latitude. As with our FLIGHTS data, we discarded rows with missing values, as well as rows with outlier longitude or latitude values (which did not correspond to real locations). The taxi data stressed our algorithm’s ability to deal with low-selectivity candidates, since more than 3000 candidates have fewer than 10 total datapoints.

POLICE Dataset. Our POLICE dataset summarizes more than 8 million police road stops in Washington state [162]. We extracted attributes for county, two gender attributes, two race attributes,

Query	Z ($ V_Z $)	X ($ V_X $)	k	target
F- q_1	Origin (347)	DepartureHour (24)	10	Chicago ORD
F- q_2	Origin (347)	DepartureHour (24)	10	Appleton ATW
F- q_3	Origin (347)	DayOfWeek (7)	5	$1/[4, 8, 8, 8, 8, 8]$
F- q_4	Origin (347)	Dest (351)	10	closest $\bar{\mathbf{r}}_i$ to uniform
T- q_1	Location (7641)	HourOfDay (24)	10	closest $\bar{\mathbf{r}}_i$ to uniform
T- q_2	Location (7641)	MonthOfYear (12)	10	closest $\bar{\mathbf{r}}_i$ to uniform
P- q_1	RoadID (210)	ContrabandFound (2)	10	closest $\bar{\mathbf{r}}_i$ to uniform
P- q_2	RoadID (210)	OfficerRace (5)	10	closest $\bar{\mathbf{r}}_i$ to uniform
P- q_3	Violation (2110)	DriverGender (2)	5	closest $\bar{\mathbf{r}}_i$ to uniform

Table 5.3: Summary of queries used to evaluate FASTMATCH

road number, violation type, stop outcome, whether a search was conducted, and whether contraband was found, for 10 attributes total.

Queries and Query Format. We evaluate several queries on each dataset, whose templates are summarized in Table 5.3. We had four queries on FLIGHTS, FLIGHTS-q1-q4, two on TAXI, TAXI-q1-q2, and three on POLICE, POLICE-q1-q3. For simplicity, in all queries we test, the x-axis is generated by grouping over a single attribute (denoted by “X” in Table 5.3), and the different candidates are likewise generated by grouping over a single (different) attribute (signified by “Z”). For each query, the visual target was chosen to correspond with the closest distribution (under ℓ_1) to uniform, out of all histograms generated via the query’s template, except for q1, q2, and q3 of FLIGHTS. Our queries spanned a number of interesting dimensions: (i) *frequently-appearing top-k candidates*: FLIGHTS-q1, POLICE-q1 and q2, (ii) *rarely-appearing top-k candidates*: FLIGHTS-q2 and q3, (iii) *high-cardinality candidate attribute Z*: TAXI-q1 and q2 ($|V_Z| = 7641$), POLICE-q3 ($|V_Z| = 2110$), and (iv): *high-cardinality grouping attribute X*: FLIGHTS-q4 ($|V_X| = 351$). The taxi queries in particular stressed our algorithm’s ability to deal with low-selectivity candidates, since more than 3000 locations have fewer than 10 total datapoints.

5.5.2 Experimental Setup

Approaches. We compare FASTMATCH against a number of less sophisticated approaches that provide the same guarantee as FASTMATCH. All approaches are parametrized by a minimum selectivity threshold σ , and all approaches except Scan are additionally parametrized by ε and δ and satisfy Guarantees 5.1 and 5.2 with probability greater than $1 - \delta$.

- SyncMatch($\varepsilon, \delta, \sigma$). This approach uses FASTMATCH, but the AnyActive block selection policy is applied without lookahead, synchronously and for each individual block. *By comparing*

this method with FASTMATCH, we quantify how much benefit we may ascribe to the lookahead technique.

- **ScanMatch($\varepsilon, \delta, \sigma$).** This approach uses FASTMATCH, but without the AnyActive block selection policy. Instead, no blocks are pruned: it scans through each block in a sequential fashion until the statistics engine reports that HistSim’s termination criterion holds. *By comparing this with SyncMatch, we quantify how much benefit we may ascribe to AnyActive block selection.*
- **Scan(σ).** This approach is a simple heap scan over the entire dataset and always returns correct results, trivially satisfying **Guarantees 5.1** and **5.2**. It exactly prunes candidates with selectivity below σ . *By comparing Scan with our above approximate approaches, we quantify how much benefit we may ascribe to the use of approximation.*

Environment. Experiments were run on single Intel Xeon E5-2630 node with 125 GiB of RAM and with 8 physical cores (16 logical) each running at 2.40 GHz, although we use at most 2 logical cores to run FASTMATCH components. The Level 1, Level 2, and Level 3 CPU cache sizes are, respectively: 512 KiB, 2048 KiB, and 20480 KiB. We ran Linux with kernel version 2.6.32. We report results for data stored in-memory, since the cost of main memory has decreased to the point that most interactive workloads can be performed entirely in-core. Each run of FASTMATCH or any other approximate approach was started from a random position in the shuffled data. We report both wall clock times and accuracy as the average across 30 runs with identical parameters, with the exception of Scan, whose wall clock times we report as the average over 5 runs. Where applicable, we used default settings of $m = 5 \cdot 10^5$, $\delta = 0.01$, $\varepsilon = 0.04$, $\sigma = 0.0008$, and $\text{lookahead} = 1024$. We set the block size for each column to 600 bytes, which we found to perform well; our results are not too sensitive to this choice.

5.5.3 Metrics

We use several metrics to compare FASTMATCH against our baselines in order to test two hypotheses: one, that FASTMATCH does indeed provide accurate answers, and two, that the system architecture developed in **Section 5.4** does indeed allow for earlier termination while satisfying the separation and reconstruction guarantees.

Wall-Clock Time. Our primary metric evaluates the end-to-end time of our approximate approaches that are variants of FASTMATCH, as well as a scan-based baseline.

Satisfaction of **Guarantees 5.1 and **5.2**.** Our δ parameter ($\delta = 0.01$), serves as an upper bound on the probability that either of these guarantees are violated. If this bound were tight, we would expect to see about one run in every hundred fail to satisfy our guarantees. We therefore count the number of times our guarantees are violated relative to the number of queries performed.

Query	Avg Speedup over Scan (raw time in (s))			
	Scan(s)	ScanMatch	SyncMatch	FASTMATCH
F-q1	12.26	27.74× (0.44)	25.53× (0.48)	37.52× (0.33)
F-q2	12.29	3.17× (3.87)	2.73× (4.51)	10.11× (1.21)
F-q3	11.62	4.76× (2.44)	3.14× (3.70)	8.72× (1.33)
F-q4	13.97	5.93× (2.36)	5.76× (2.43)	8.15× (1.71)
T-q1	13.09	4.89× (2.68)	0.32× (40.95)	15.93× (0.82)
T-q2	13.09	6.48× (2.02)	0.37× (35.60)	17.38× (0.75)
P-q1	8.57	5.72× (1.50)	5.14× (1.67)	13.34× (0.64)
P-q2	8.49	14.31× (0.59)	15.48× (0.55)	36.11× (0.24)
P-q3	8.65	9.25× (0.93)	1.53× (5.66)	33.26× (0.26)

Table 5.4: Summary of average query speedups and latencies

Total Relative Error in Visual Distance. In some situations, there may be several candidate histograms that are quite close to the analyst-supplied target, and choosing any one of them to be among the k returned to the analyst would be a good choice. We define the *total relative error in visual distance* (denoted by Δ_d) between the k candidates returned by FASTMATCH and the true k closest visualizations as: $\Delta_d(M, M^*, \mathbf{q}) = \frac{\sum_{i \in M} d(\mathbf{r}_i, \mathbf{q}) - \sum_{j \in M^*} d(\mathbf{r}_j^*, \mathbf{q})}{\sum_{j \in M^*} d(\mathbf{r}_j^*, \mathbf{q})}$. Note that here, M^* is computed by Scan and only considers candidates meeting the selectivity threshold. Since FASTMATCH and our other approximate variants have no recall requirements with respect to identifying low-selectivity candidates (they only have precision requirements), it is possible for $\Delta_d < 0$.

5.5.4 Empirical Results

Speedups and Error of FASTMATCH versus others.

Summary. All FASTMATCH variants we tested show significant speedups over Scan for at least one query, but only FASTMATCH shows consistently excellent performance, typically beating other approaches and bringing latencies for all queries near interactive levels; with an overall speedup ranging between **8×** and **35×** over Scan. Further, the output of **FASTMATCH** and **all approximate variants** satisfied **Guarantees 5.1** and **5.2** across all runs for all queries.

Average run times of FASTMATCH and other approaches, for all queries as well as speedups over Scan, are summarized in Table 5.4. We used default settings for all runs. The reported speedups are the ratio of the average wall time of Scan with the average wall time of each approach considered. Scan was generally slower than approximate approaches because it had to examine all the data. Then, we typically observed that ScanMatch and SyncMatch were pretty evenly matched, with ScanMatch

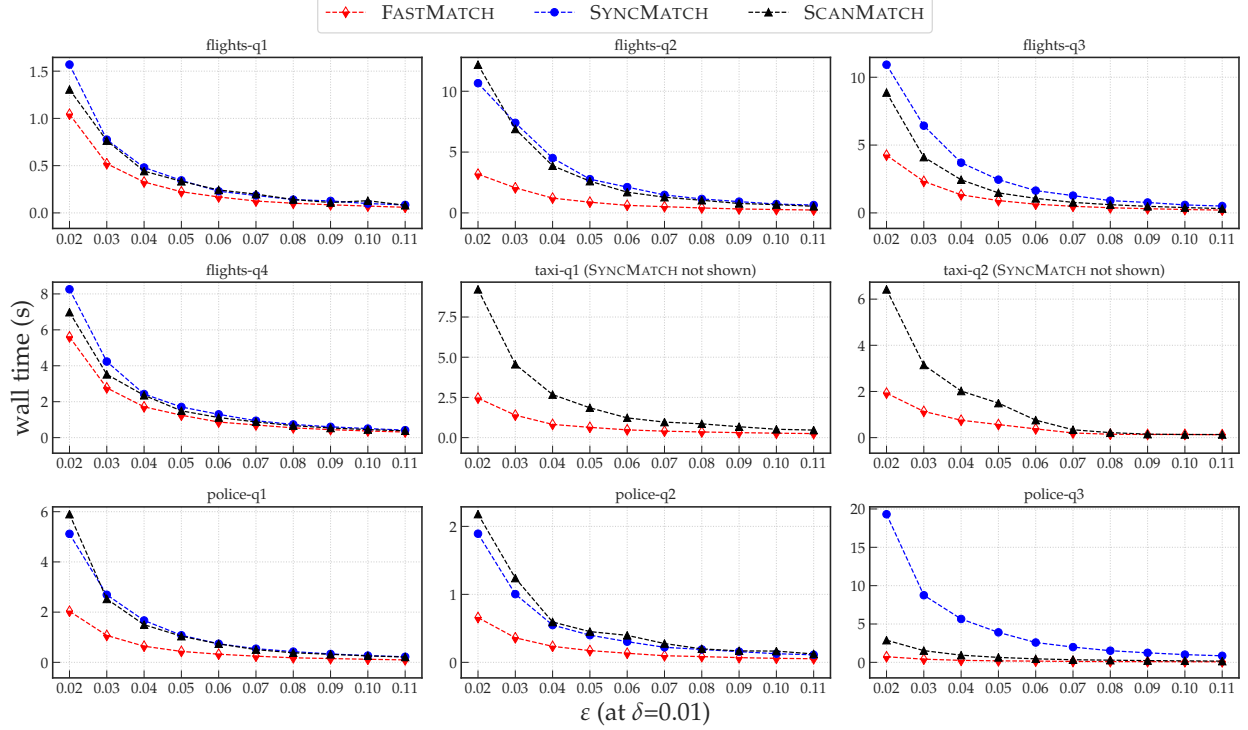


Figure 5.9: Effect of ε on query latency

usually performing slightly better, except in some pathological cases where it performed very poorly due to poor cache usage. FASTMATCH had better performance than either SyncMatch or ScanMatch, thanks to lookahead paired with AnyActive block selection. Overall, we observed that each of FASTMATCH’s key innovations: the termination criterion, the block selection, and lookahead, all led to substantial performance improvements, with an overall speedup of up to **35×** over Scan.

Queries with high candidate cardinality (TAXI-q*, POLICE-q3), displayed particularly interesting performance differences. For these, FASTMATCH shows greatly improved performance over ScanMatch. It also scales much better to the large number of candidates than SyncMatch, which performs extremely poorly due to poor cache utilization and takes around $3\times$ longer than a simple non-approximate Scan. In this case, the lookahead technique of FASTMATCH is necessary to reap the benefits of AnyActive block selection.

Additionally, we found that the output of *FASTMATCH* and all approximate variants satisfied *Guarantees 5.1 and 5.2 across all runs for all queries*. This suggests that the parameter δ may be a loose upper bound for the actual failure probability of FASTMATCH.

Effect of varying ε .

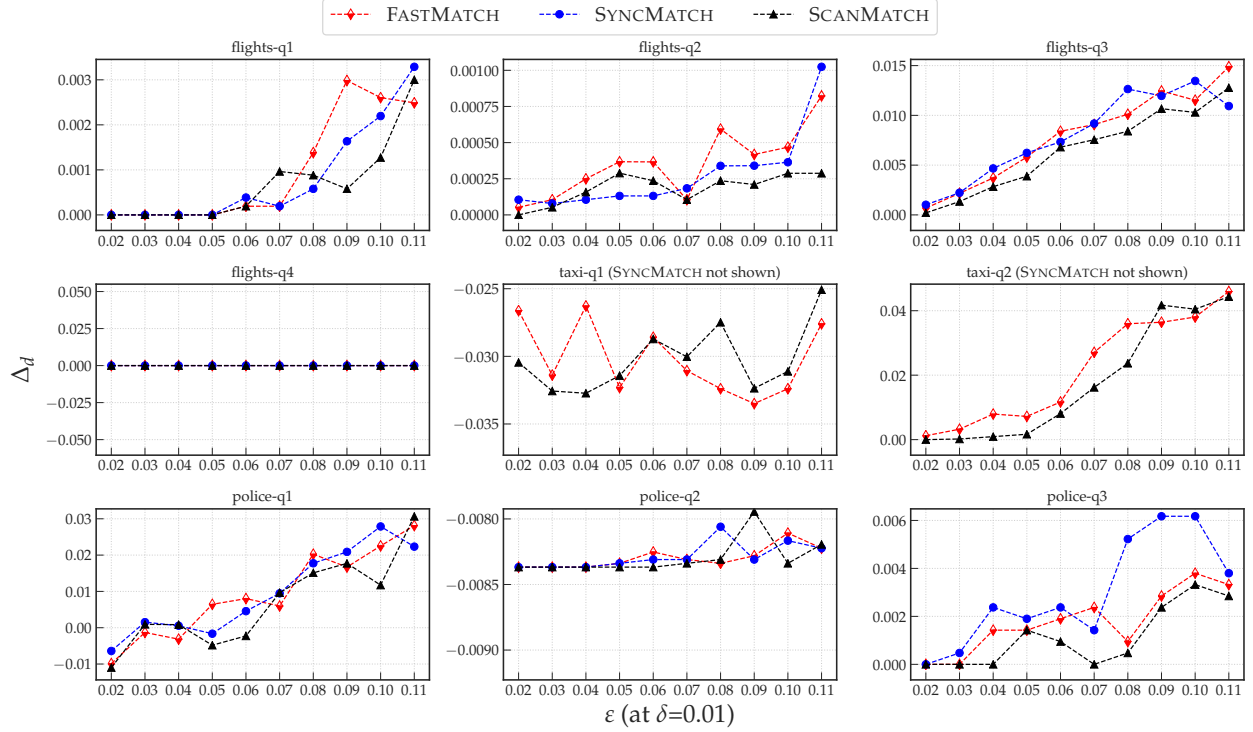


Figure 5.10: Effect of ε on Δ_d

Summary. In almost all cases, increasing the tolerance parameter ε leads to reduced runtime and accuracy, but **on average, Δ_d was never more than 5% larger than optimal for any query, even for the largest values of ε used.**

Figures 5.9 and 5.10 depict the effect of varying ε on the wall clock time and on Δ_d , respectively, using $\delta = 0.01$ and lookahead = 1024, averaged over 30 runs for each value of ε . Because of the extremely poor performance of SyncMatch on the TAXI queries, we omit it from both figures.

In general, as we increased ε , wall clock time decreased and Δ_d increased. In some cases, ScanMatch latencies matched that of Scan until we made ε large enough. This sometimes happened when it needed more refined estimates of the (relatively infrequent) top- k candidates, which it achieved by scanning most of the data, picking up lots of superfluous (in terms of achieving safe termination) tuples along the way.

Effect of varying lookahead.

Summary. When the number of candidates $|V_Z|$ is not large, performance is relatively stable as lookahead varies. For large $|V_Z|$, more lookahead helps performance, but is not crucial.

For most queries, we found that latency was relatively robust to changes in lookahead. Figure 5.11 depicts this effect. The queries with high candidate cardinalities (TAXI-q*, POLICE-q3) were the

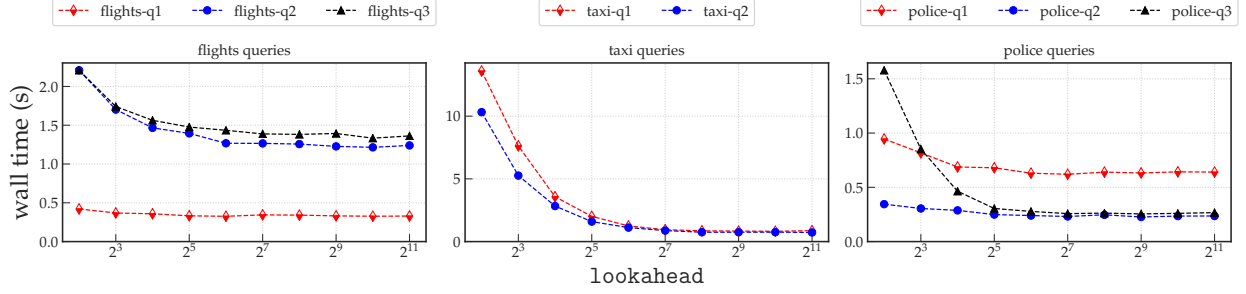


Figure 5.11: Effect of varying lookahead

exceptions. For these queries, larger `lookahead` values led to increased utilization at all levels of CPU cache. Past a certain point, however, the performance gains were minor. Overall, we found the default value of 1024 to be acceptable in all circumstances.

Effect of varying δ . In general, we found that increasing δ led to slight decreases in wall clock time, leaving accuracy (in terms of Δ_d) more or less constant. We believe this behavior is inherited from our bound in [Theorem 5.1](#), which is not sensitive to changes in δ . [Figure 5.12](#) shows the effect of varying δ on wall clock time. For the values of δ we tried, we did not observe any meaningful trends in Δ_d and have omitted the plot.

When approximation performs poorly. In order to achieve the competitive results presented in this section, the initial pruning of low-selectivity candidates during stage 1 of HistSim ended up being critical for good performance. With a selectivity threshold of $\sigma = 0$, stages 2 and 3 of HistSim are forced to consider many extremely rare candidates. For example, in the taxi queries, nearly half of candidates have fewer than 10 corresponding datapoints. In this case, ScanMatch performs the best (essentially performing a Scan with a slight amount of additional overhead), but it (necessarily) fails to take enough samples to establish [Guarantees 5.1](#) and [5.2](#). SyncMatch and FASTMATCH likewise fail to establish guarantees, but additionally have the issue of being forced to consider many rare candidates while employing AnyActive block selection, which can slow down query processing by a factor of 100× or more.

Comparing results for ℓ_1 and ℓ_2 metrics. So far, we have not validated our choice of distance metric (normalized ℓ_1); prior work has shown that normalized ℓ_2 is suitable for assessing the “visual” similarity of visualizations [\[138\]](#), so here, we compare our top-k with the top-k using the normalized ℓ_2 metric, for the FLIGHTS queries. In brief, we found that the relative difference in the total ℓ_1 distance of the top-k using the two metrics never exceeded 4% for any query, and that roughly 75% of the top-k candidates were common across the two metrics. Thus, ℓ_1 can serve as a suitable replacement for ℓ_2 , while further benefiting from the advantages we described in [Section 5.2](#). [Table 5.5](#) summarizes our full results.

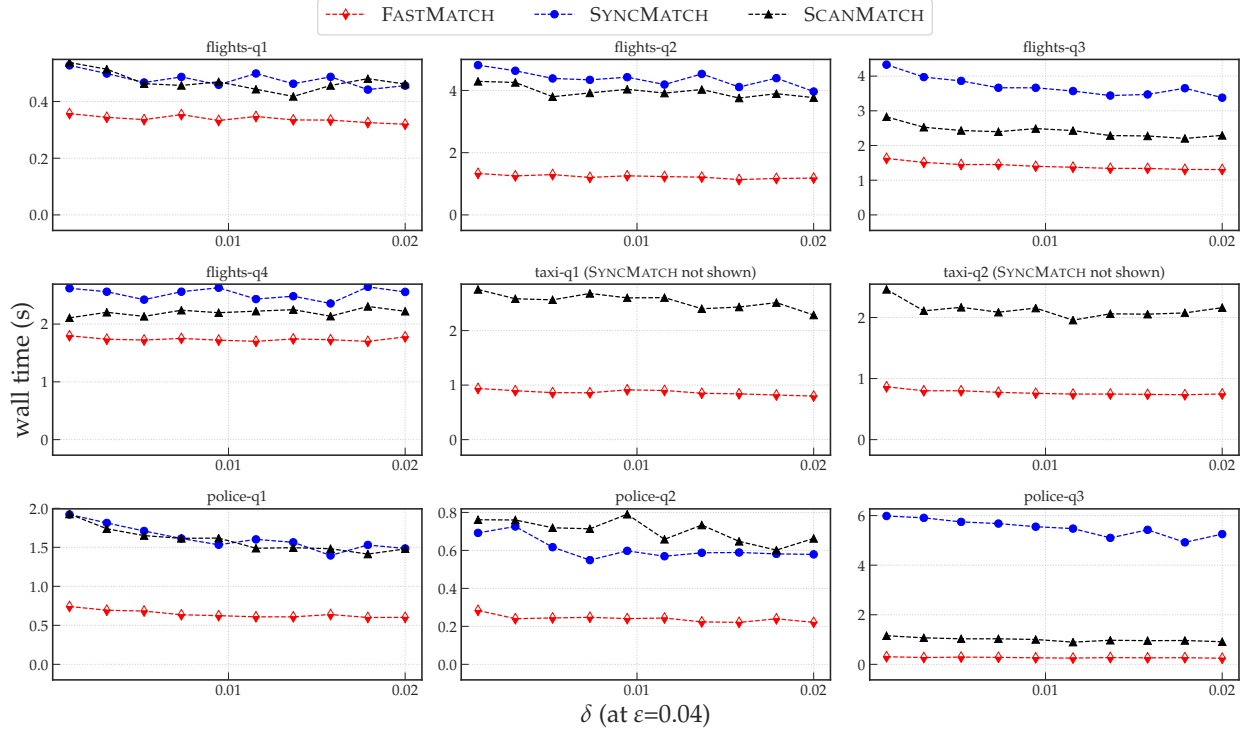


Figure 5.12: Effect of δ on wall clock time

Query	$\frac{ M^*(\ell_1) \cap M^*(\ell_2) }{k}$	Relative distance difference
FLIGHTS- q_1	0.9	0.01
FLIGHTS- q_2	0.7	0.04
FLIGHTS- q_3	0.6	0.03
FLIGHTS- q_4	0.8	0.01

Table 5.5: Comparison of top-closest histograms for ℓ_1 and ℓ_2 metrics.

5.6 SUMMARY

We developed sampling-based strategies for rapidly identifying the top- k histograms that are closest to a target. We designed a general algorithm, HistSim, that provides a principled framework to facilitate this search, with theoretical guarantees. We showed how the systems-level optimizations present in our FASTMATCH architecture are crucial for achieving near-interactive latencies consistently, leading to speedups ranging from $8\times$ to $35\times$ over baselines.

In the next chapter, we extend the FASTMATCH architecture to support more general report generation via techniques for approximating queries involving aggregates.

CHAPTER 6: SAFE APPROXIMATION FOR REPORT GENERATION

In this chapter, we extend the FASTMATCH architecture to develop FASTFRAME, a more general system that uses guaranteed error bounds to facilitate queries involving aggregates, with downstream applications to report generation. We begin with motivation, then describe several issues that existing error bounding techniques suffer from, before describing the data-aware mitigations that FASTFRAME employs. After further elaborating on the system architecture from Chapter 5, we perform an empirical study that evaluates FASTFRAME along the axes of safety and interactivity, again on several real datasets.

6.1 MOTIVATION

Primitives for aggregation like AVG, SUM, and COUNT are key to making sense of and drawing insights from large volumes of data, powering applications in OLAP, exploratory data analysis, and visual analytics. Accelerating their computation is therefore of great importance. Approximate Query Processing (AQP) is commonly used to accelerate computation of these aggregates by estimating them on a subset or sample of the full data. Reasoning about the error of the estimates as introduced by approximation is crucial: consumers of approximate answers—ranging from human decision makers to automated processes—rely on confidence intervals (CIs) or error bounds as the foundation for understanding the quality of the approximate answer. Therefore, many AQP techniques come with CIs to allow for more confident or informed decisions made using approximate estimates.

Error bounding, or CI computation techniques take a confidence parameter $\delta \in [0, 1]$, with the semantics that the returned intervals $[g_\ell, g_r]$ fail to enclose the true aggregate g^* at most δ of the time. One can tune δ to be as small as needed at the cost of requiring more samples to achieve the same interval width ($g_r - g_\ell$). Likewise, for a given δ , taking more samples typically causes the error bounding procedure to return a narrower confidence interval. Since δ is typically small, we use the phrase “with high probability” (w.h.p.) as shorthand for “with probability greater than $(1 - \delta)$ ”. CI computation techniques need to satisfy two goals: **(i) compactness:** *by minimizing the interval width $g_r - g_\ell$* , and **(ii) correctness:** *by ensuring that $g^* \in [g_\ell, g_r]$ with high probability*. However, achieving both compactness and correctness simultaneously is difficult.

In general, conservative methods such as those based on Hoeffding’s inequality [155] or on the Hoeffding-Serfling inequality [163] rely on a-priori knowledge of *range bounds* a and b between which the data fall (typically inferred during data loading). Although they achieve the correctness goal of error bounders, when used for AVG, the CI width for Hoeffding-based error bounders scales

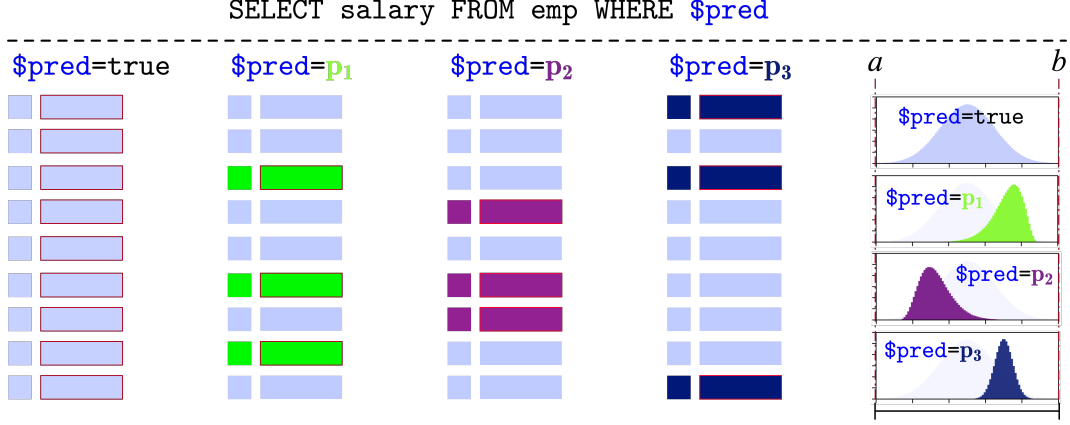


Figure 6.1: Few points may lie near the range bounds a and b , and with filters applied, the true range could be significantly smaller than $(b - a)$.

with the range size $(b - a)$, creating at least two major issues in the context of a relational database, illustrated in Figure 6.1. (i) *First, the presence of a very few outliers can significantly widen the range* $[a, b]$ (and therefore the CI width), even though most of the data may lie in a much smaller range. In Figure 6.1, we see that even though the range of salaries is $b - a$ when $\$pred = \text{true}$, most of the data is concentrated in the center of the range. (ii) *Second, predicates and groupings may be applied during data exploration, so that the filtered data lies in a smaller range than* $[a, b]$; in Figure 6.1, with $\$pred = p_i$, we see that the range of filtered salaries is much smaller than even the $\$pred = \text{true}$ case. However, direct application of Hoeffding-based methods do not account for the tighter range of the filtered data, instead treating the sampled tuples as if they were taken from the original (unfiltered) data.

Thus, the focus of this chapter is to more deeply understand the issues that traditional conservative error bounders suffer from, propose mitigations, and then extend our system architecture from Chapter 5 to leverage these mitigations for interactive latencies while preserving strong correctness guarantees.

6.2 DBMS ERROR BOUND INTEGRATION

In this section, we first describe applications of confidence intervals for facilitating query processing in a database system (§6.2.1). Next, we survey methods for computing error bounds with guarantees applicable to DBMS aggregates (§6.2.2) identify their shortcomings (§6.2.3) and conclude with a formal problem statement (§6.2.4).

Symbols / Terms	Descriptions
\mathcal{D}, N, S, c, m	Dataset, num. points in dataset (i.e. $ \mathcal{D} $), sample, num. points taken (for c) or desired (for m) in sample (i.e. $ S $)
$g^*, \hat{g}, g_\ell, g_r$	True aggregate, estimate, error bounds
$a, b, \sigma^2, \hat{\sigma}^2, \delta, \varepsilon$	Range bounds, variance, empirical variance, error probability upper bound, error
F, \hat{F}, L, U	True / empirical CDF, lower and upper bounds on true CDF
Lbound, Rbound	Confidence lower (resp. upper) bounding routines parameterized on a, b, N , and other sample state (see §6.2.2).
SSI, PMA, PHOS	Sample-size-independent, pessimistic mass allocation, phantom outlier sensitivity

Table 6.1: Glossary of terms and notation used in Chapter 6

```
SELECT Origin, AVG(DepDelay) FROM flights
GROUP BY Origin HAVING AVG(DepDelay) < 0
```

Figure 6.2: Origin airports with negative average delay. In this query, the AVG aggregates are consumed both by the user and by the system.

6.2.1 DBMS CI Applications

Consider the query in Figure 6.2. In this query, AVG aggregates are both *displayed* as output in the query results, and are also used to *filter* the set of tuples in the output. This reflects two major applications of confidence intervals in a DBMS setting: CIs that are *explicitly used* downstream, i.e., by an analyst, or CIs that are *implicitly used* by automated processes.

Explicit Use of Downstream CIs. When approximating aggregates in a DBMS, CIs can be included in the output displayed to users. For example, the AVG aggregates belonging to the groups output by the query in Figure 6.2 are augmented with CIs and included in the output. Such CIs help users *reason about uncertainty in approximate answers* during analysis [2, 72].

Implicit Use of Downstream CIs. Confidence intervals have been applied towards various downstream applications, for example, to enable early stopping. Example applications include high-level accuracy contracts [4, 77] (i.e., guaranteeing query results are within ε of the correct), ranking query results [6], and bounding relative error [24]. In all cases, the user need not ever observe the interval: the goal is to provide *early stopping* while ensuring *correct results*. We consider these applications later in our experiments in Section 6.5.

Goal. In this chapter, we are primarily concerned with enabling CI *compactness* (to reduce query latency) without sacrificing CI *correctness* (thereby ensuring safety), for both explicit and implicit

applications of CIs. *The major goal is therefore to develop CI techniques that are as tight as possible, while always enclosing the quantity in question.* Throughout this section and [Section 6.3](#), we will focus our discussion on CIs for AVG aggregates; we will cover SUM and COUNT aggregates in [Section 6.4](#).

6.2.2 Computing CIs in a DBMS

We now describe methods for computing error bounds with accuracy guarantees in a database system, along with any assumptions required. Relevant notation is summarized in [Table 6.1](#). We begin by defining error bounders, bounds, and confidence intervals.

Definition 6.1 *[(1 − δ) error bounders and bounds]. A procedure P that returns error bounds $[g_\ell, g_r]$ for some aggregate g^* given a sample is a (1 − δ) error bounder if, across all possible samples, $\mathbb{P}(g^* \notin [g_\ell, g_r]) < \delta$. $[g_\ell, g_r]$ is called the (1 − δ) confidence interval for g^* , and g_ℓ and g_r are collectively referred to as (1 − δ) error or confidence bounds.*

In contrast with asymptotic error bounders that only satisfy $\mathbb{P}(g^* \notin [g_\ell, g_r]) \approx \delta$ for large-enough sample sizes, the (1 − δ) error bounders from [Definition 6.1](#) always satisfy $\mathbb{P}(g^* \notin [g_\ell, g_r]) < \delta$ for any sample size, so we call them *sample-size-independent* (SSI).

Assumptions Applicable to Data in a DBMS

In the case of AVG aggregates, all error bounding procedures require some prior knowledge about the data over which they operate – otherwise, outliers can have arbitrarily strong effects on the aggregate in question. Weaker assumptions are more general, but typically yield more conservative bounds.

In this chapter, we make two assumptions about the data \mathcal{D} over which queries operate: first, that every datapoint $x \in \mathcal{D}$ lies in some interval $[a, b]$; second, that datapoints can be effectively sampled *without replacement* from \mathcal{D} , as in the previous chapter. We now discuss these assumptions in the context of prior work and show that they can be implemented effectively within real systems.

Known Range Bounds. As in prior work [\[2\]](#), we assume that the database catalog maintains *range bounds* a and b for the MIN and MAX of each continuous column, inferred, for example, during data loading. (Note that we do not require $[a, b] = [\text{MIN}, \text{MAX}]$, but only that $[a, b] \supseteq [\text{MIN}, \text{MAX}]$.) These assumptions are more applicable in the context of a database as compared with stronger distributional assumptions (e.g., that the data are normal or that they obey a tighter sub-Gaussian parameter than that implied by the range bounds [\[164\]](#)) and can be easily maintained in the case of insertions. We refer to bounders that assume knowledge of a and b as *range-based error bounders*.

throughout this chapter. Furthermore, it is possible to leverage the range assumption even in the case of aggregates involving arbitrary expressions over multiple columns by first solving an optimization problem for derived range bounds a' and b' that enclose the transformed data.

Sampling Without Replacement. Estimates for AVG aggregates generally converge faster for samples taken without replacement than samples taken with replacement [156, 163]. In the context of a DBMS, sampling with replacement has traditionally been considered easier than sampling without replacement, since the system does not need to “remember” the samples already taken [6, 165]. Sampling as traditionally implemented, however, also has poor locality properties, as nearly every read operation results in a cache miss. Another approach taken in prior work [157, 158, 159] is to materialize samples ahead-of-time by performing a single up-front shuffle of the entire relation, so that sampling without replacement can be implemented via a scan of the data *regardless* of any applied filters or other transformations. As we leverage the same architecture from Chapter 5, we opt for this approach.

State for DBMS Error Bounds

OLAP queries must operate over many tuples, so it is desirable that aggregations and their error bounders maintain small of memory footprints as possible as new tuples are examined, although we will see in Section 6.2.2 that some bounders must maintain state which grows with the number of tuples examined. To better understand implementation details for error bounders within the context of a DBMS, we present error bounders in terms of the following interface:

- ① `init_state()`: Initializes state needed for error bounds.
- ② `update_state(S, v)`: Given the current state S and a newly-seen value v , compute state S' .
- ③ `Lbound(S, a, b, N, δ)`: Return a confidence lower bound for a sample whose relevant statistics are captured in state S , assuming the sample came from a finite dataset \mathcal{D} of N values in $[a, b]$. The probability of a sample that causes Lbound to return a value greater than $\text{AVG}(\mathcal{D})$ is $< \delta$.
- ④ `Rbound(S, a, b, N, δ)`: Symmetric to Lbound for the confidence upper bound. Can typically be implemented in terms of Lbound after a suitable transformation of S .

The state S captures information such as the count of tuples examined and the current running average, as well as anything else required by Lbound and Rbound. The state initialization and update logic is analogous to state maintenance logic for aggregate functions as implemented in existing commercial database systems [166, 167, 168, 169].

Note that both Lbound and Rbound depend on the range bounds a and b , as well as the data size N (allowing for tighter bounds when sampling without replacement).

Algorithm 6.1: Hoeffding-Serfling error bounder [163]

```
1 function init_state() ❶
2   return { $m$ : 0,  $\hat{g}$ : 0};

3 function update_state( $S, v$ ) ❷
4    $m' \leftarrow S.m + 1$ ;
5    $\hat{g}' \leftarrow S.\hat{g} + (v - S.\hat{g})/m'$ ;
6   return { $m$ :  $m'$ ,  $\hat{g}$ :  $\hat{g}'$ };

7 function Lbound( $S, a, b, N, \delta$ ) [163] ❸
8    $\varepsilon \leftarrow (b - a) \cdot \sqrt{\frac{\log(1/\delta)}{2 \cdot S.m} \cdot (1 - \frac{S.m-1}{N})}$ ;
9   return  $S.\hat{g} - \varepsilon$ ;

10 function Rbound( $S, a, b, N, \delta$ ) ❹
11    $S.\hat{g} \leftarrow (a + b) - S.\hat{g}$ ;
12   return  $(a + b) - \text{Lbound}(S, a, b, N, \delta)$ ;
```

Error Bounds for Finite and Bounded Data

In this section, we review some techniques for computing confidence intervals that leverage only the assumptions discussed previously: that samples are taken without-replacement from data bounded in some a priori-known range $[a, b]$. Our goal is not to be exhaustive but representative, drawing attention to previous applications in the DB literature (and lack thereof).

Hoeffding-Serfling-based Bounder. An error bounder based on the Hoeffding-Serfling inequality [163] computes CIs whose widths depend only on the range $(b - a)$ and the number of samples m , and that have size $\mathcal{O}((b - a)/\sqrt{m})$ (if we ignore the sampling fraction term). While asymptotically optimal for worst-case data distributed with half of the points at a and the other half at b , it is needlessly wide in practice, when few points occur near a or b . An implementation of this bounder in terms of our interface from Section 6.2.2 is given in Algorithm 6.1. We give a statement of the Hoeffding-Serfling inequality and derive the corresponding error bounder.

Lemma 6.1 (Hoeffding-Serfling Inequality [163]). *Let $\mathcal{D} = x_1, \dots, x_N$ be a set of N values in $[a, b]$ with average value $\text{AVG}(\mathcal{D}) = \mu$. Let X_1, \dots, X_N be a sequence of random variables drawn from \mathcal{D} without replacement. For every $1 \leq m \leq N$ and $\varepsilon > 0$,*

$$\mathbb{P}\left(\max_{1 \leq k \leq m} \frac{\sum_{t=1}^k (X_t - \mu)}{N - k} \geq \frac{m\varepsilon}{N - m}\right) \leq \delta \quad (6.1)$$

Algorithm 6.2: Empirical Bernstein-Serfling err. bounder [156]

```

1 function init_state() ❶
2   return { $m$ : 0,  $\hat{g}$ : 0,  $M_2$ : 0};

3 function update_state( $S, v$ ) ❷
4    $m' \leftarrow S.m + 1$ ;
5    $\hat{g}' \leftarrow S.\hat{g} + (v - S.\hat{g})/m'$ ;
6    $M'_2 \leftarrow S.M_2 + v^2$ ;
7   return { $m$ :  $m'$ ,  $\hat{g}$ :  $\hat{g}'$ ,  $M_2$ :  $M'_2$ };

8 function Lbound( $S, a, b, N, \delta$ ) [156] ❸
9    $\kappa \leftarrow 7/3 + 3/\sqrt{2}$ ;
10   $\rho \leftarrow \mathbb{I}\{S.m \leq N/2\} \cdot (1 - \frac{S.m-1}{N})$ ;
11   $\rho \leftarrow \rho + \mathbb{I}\{S.m > N/2\} \cdot ((1 - \frac{S.m}{N}) \cdot (1 + \frac{1}{S.m}))$ ;
12   $\varepsilon \leftarrow \sqrt{S.M_2/S.m - S.\hat{g}^2} \cdot \sqrt{\frac{2\rho \cdot \log(5/\delta)}{S.m}} + \kappa \cdot (b - a) \cdot \frac{\log(5/\delta)}{S.m}$ ;
13  return  $S.\hat{g} - \varepsilon$ ;

14 function Rbound( $S, a, b, N, \delta$ ) ❹
15   $S.\hat{g} \leftarrow (a + b) - S.\hat{g}$ ;
16  return  $(a + b) - \text{Lbound}(S, a, b, N, \delta)$ ;

```

where

$$\delta = \exp\left(-\frac{2m\varepsilon^2}{(1 - \frac{m-1}{N})(b-a)^2}\right) \quad (6.2)$$

By focusing on $k = m$ and inverting the probability expression, we may compute a $1 - \delta$ lower confidence bound as

$$\frac{1}{m} \sum_{t=1}^m X_t - (b-a) \sqrt{\frac{(1 - \frac{m-1}{N})(\log \frac{1}{\delta})}{2m}} \quad (6.3)$$

and likewise for an upper confidence bound (replacing “−” with “+”), so that $(1 - \frac{\delta}{2})$ lower and upper confidence bounds may be combined to yield a $(1 - \delta)$ confidence interval (via a union bound).

Empirical Bernstein-Serfling-based Bounder. A concentration inequality for sampling without replacement given in [156], the *Bernstein-Serfling* inequality assumes knowledge of both $(b - a)$ and $\text{VAR}(\mathcal{D}) = \sigma^2 = \frac{1}{N} \sum_{x \in \mathcal{D}} (x - \text{AVG}(\mathcal{D}))^2$. We defer a statement of the full result. Here we note that inverting the inequality gives error bounds as

$$\frac{1}{m} \sum_{t=1}^m X_t \pm \mathcal{O}(\sigma/\sqrt{m} + (b-a)/m) \quad (6.4)$$

if we again ignore the sampling fraction term. Comparing these error bounds to those of Hoeffding-Serfling, which has widths of size $\mathcal{O}((b-a)/\sqrt{m})$ (again ignoring the sampling fraction), we see that error bounds derived from the Bernstein-Serfling inequality can be significantly tighter when σ is small compared to $(b-a)$.

Knowledge of $\text{VAR}(\mathcal{D})$ typically cannot be assumed in a setting where $\text{AVG}(\mathcal{D})$ is unknown. Fortunately, there also exists an *empirical* variant of the Bernstein-Serfling inequality (also given in [156], like the non-empirical variant). The analysis for the empirical Bernstein-Serfling inequality proceeds by augmenting the analysis for the non-empirical variant with a concentration inequality relating the estimator $\hat{\sigma}^2 = \frac{1}{m} \sum_{t=1}^m (X_t - \bar{X})^2$ to $\text{VAR}(\mathcal{D})$. We again deferring the full statement. This yields $(1 - \delta)$ error bounds given by

$$\frac{1}{m} \sum_{t=1}^m X_t \pm \mathcal{O}(\hat{\sigma}/\sqrt{m} + (b-a)/m) \quad (6.5)$$

Note that these error bounds differ from the those of the non-empirical variant only in that σ is replaced by $\hat{\sigma}$ (modulo slightly worse constants hidden by the asymptotic notation). Although $\hat{\sigma}$ is a random quantity, it concentrates near σ , so that an error bound based on the empirical Bernstein-Serfling bound returns bounds of asymptotically the same width as those returned by an error bound based on the non-empirical variant and with full access to σ^2 , w.h.p. [Algorithm 6.2](#) gives an implementation of an empirical Bernstein-Serfling-based error bound in terms of our interface from [Section 6.2.2](#). Note that [Algorithm 6.2](#) as presented shows computation of the sample variance in terms of the second moment $M_2 = \sum v^2$ for the sake of exposition; a real implementation might use a more numerically stable one-pass algorithm for the variance [170, 171, 172].

Anderson/DKW-based Bounder. Anderson described a way to compute distribution-free / nonparametric error bounds for the mean given error bounds for the cumulative distribution function (CDF) in [174]. Denoting the true and empirical CDF for some distribution supported on $[a, b]$ with F and \hat{F} , respectively, Anderson showed how to use high-probability bounds α and β such that

$$\hat{F} - \alpha \leq F \leq \hat{F} + \beta \quad (6.6)$$

to get high-probability bounds on the mean of F . To see how, recall the following identity:

Lemma 6.2. *Consider a CDF F supported on $[a, b]$. Then the mean μ of the distribution corresponding to F satisfies*

$$\mu = b - \int_a^b F(x) dx \quad (6.7)$$

Algorithm 6.3: Anderson/DKW error bounder [173, 174, 175]

```

1 function init_state() ❶
2   return {}

3 function update_state( $S, v$ ) ❷
4   return  $S \cup \{v\}$ 

5 function Lbound( $S, a, b, N, \delta$ ) ❸
6    $\varepsilon \leftarrow \sqrt{\frac{\log(1/\delta)}{2 \cdot |S|}};$ 
7    $\hat{F} \leftarrow$  empirical CDF based on  $S$ ;
8    $S' \leftarrow \{x \in S : \hat{F}(x) \leq 1 - \varepsilon\};$ 
9   return  $\varepsilon \cdot a + (1 - \varepsilon) \cdot \text{AVG}(S')$ ;

10 function Rbound( $S, a, b, N, \delta$ ) ❹
11  return  $(a + b) - \text{Lbound}((a + b) - S, a, b, N, \delta);$ 

```

Thus, given lower and upper bounds L and U on the CDF F that satisfy $\forall x \in [a, b], L(x) \leq F(x) \leq U(x)$, error bounds around the mean may be computed as

$$\left[b - \int_a^b U(x)dx, \quad b - \int_a^b L(x)dx \right] \quad (6.8)$$

since $L \leq F \leq U$ implies $-U \leq -F \leq -L$.

Anderson used the Dvoretzky-Kiefer-Wolfowitz (DKW) inequality [173] to compute α and β . Informally, DKW states that the empirical CDF \hat{F} computed from i.i.d. samples taken from a distribution with CDF F concentrates around F everywhere:

Lemma 6.3 (DKW Inequality [173, 175]). *Let $X_1, \dots, X_m \stackrel{iid}{\sim} F$, and let \hat{F} be the empirical CDF corresponding to the sample $\{X_i\}$. Then for every $\varepsilon > 0$,*

$$\mathbb{P} \left(\sup_{t \in \text{dom}(F)} |\hat{F}(t) - F(t)| > \varepsilon \right) \leq 2 \exp(-2m\varepsilon^2) \quad (6.9)$$

The DKW inequality provides a method to obtain the values of α and β , since it implies that

$$\hat{F} - \sqrt{\frac{\log 2/\delta}{2m}} \leq F \leq \hat{F} + \sqrt{\frac{\log 2/\delta}{2m}} \quad (6.10)$$

Error Bounder	PMA	PHOS	Sampling	Memory
Hoeffding(-Serfling)	✓	✓	R* (NR)	$\mathcal{O}(1)$
Berstein(-Serfling)		✓	R* (NR)	$\mathcal{O}(1)$
Anderson/DKW	✓		R, NR	$\mathcal{O}(m)$

Table 6.2: Summary of properties exhibited by various error bounders. R = sampling with replacement, NR = without. A * indicates that the non-Serfling variant also holds for NR sampling.

with probability greater than $1 - \delta$. At the time [174] was published, however, the constant in front of the DKW inequality had not yet been proved by Massart [175], so it appears that Anderson computed α and β using a lookup table.

Although Lemma 6.3 as stated applies for sampling with replacement from an infinite population, please see Appendix A.5 for a proof that DKW still holds when X_1, \dots, X_m are drawn without replacement from a finite population of size N , for any $N > 0$, stated as the following theorem:

Theorem 6.1. *For any $N > 0$, the DKW inequality applies for sampling without replacement from a finite dataset of size N .*

The procedure just described for computing error bounds around the mean of a distribution given i.i.d. samples thus also works for computing error bounds around $\text{AVG}(\mathcal{D})$ given without-replacement samples from the finite dataset \mathcal{D} . It is presented in terms of our interface from Section 6.2.2 in Algorithm 6.3.

Applications in Prior DB Literature. To our knowledge, Hoeffding and Hoeffding-Serfling-based bounders are the only SSI bounders that have seen extensive use in the DB literature for computing error bounds for AVG [2, 6, 24, 176]. We are aware of one incorrect application of the empirical Bernstein-Serfling inequality [177] (incorrect because the procedure given in [177] continuously recomputes confidence $(1 - \delta)$ intervals as more samples are taken, so that the overall procedure is no longer guaranteed to fail with probability at most δ). Overall it is somewhat surprising that error bounders derived from the empirical Bernstein-Serfling inequality [156] have not seen more widespread usage, as they are nearly as simple to compute as those derived from the Hoeffding-Serfling inequality and typically yield error bounds that are much tighter.

6.2.3 Error Bounder Pathologies

We identify two problems that cause SSI error bounders to be *too* conservative. These pathologies, which we refer to as *pessimistic mass allocation (PMA)* and *phantom outlier sensitivity (PHOS)*, are based on simple intuitions about how error bounders should behave: namely, they should return tighter bounds when observing samples with fewer extreme values, and error lower bounds

(respectively error upper bounds) should only be looser due to potential large values (resp. small values) if such values are actually observed.

Pessimistic Mass Allocation

PMA captures the intuition that error bounders should be sensitive to the observed sample values:

Definition 6.2 [PMA]. *An error bounding procedure P exhibits pessimistic mass allocation (PMA) if there exists a dataset \mathcal{D} bounded in $[a, b]$, a value a' with $a < a' < b$, and a set $S \subseteq \mathcal{D}$ with values in $[a, a')$ such that, for $S' = \{\max(x, a') : x \in S\}$, P returns a confidence interval of the same width for both S and S' . P likewise exhibits PMA if there exists some b' with $a < b' < b$ and an S with values in $(b', b]$ such that, for $S' = \{\min(x, b') : x \in S\}$, P returns a confidence interval of the same width for both S and S' .*

That is, for an error bouncer P with PMA, we can replace the smallest (largest) elements in a sample with something larger (resp. smaller) without shrinking the width of P 's returned confidence interval. For example, for data known to lie in $[0, 1]$, P might yield an interval of the same width for both a sample split evenly between 0 and 1 as well as a sample split evenly between 0.25 and 0.75, even though the latter sample should clearly give rise to a tighter interval. Intuitively, P is overly-pessimistic about how mass in the underlying distribution from which it is sampling is allocated, despite contrary evidence observed in the sample.

Phantom Outlier Sensitivity

PHOS captures the intuition that unobserved extreme values should not affect both the lower and the upper error bounds computed by some error bouncer P :

Definition 6.3 [PHOS]. *An error bounding procedure P exhibits phantom outlier sensitivity (PHOS) if, for data falling in $[a, b]$, P 's returned confidence lower bound g_ℓ depends on the value of b , and similarly if the g_r returned by P depends on a .*

To understand PHOS intuitively, consider the case of computing a confidence lower bound. Given a sample S , the worse P “believes” S could be shifted (on average) toward larger values as compared to \mathcal{D} , the smaller of a confidence lower bound it should return. In what ways could S be shifted toward higher values? One possibility is if small elements are underrepresented in S . The other possibility, and the one we are interested in, is if large elements are overrepresented in S . For this reason, a confidence lower bound should only be affected by datapoints near the upper range bound b if it actually observes them, and the appearance of b in the computation of a confidence lower bound is a potential source of unnecessary conservativeness.

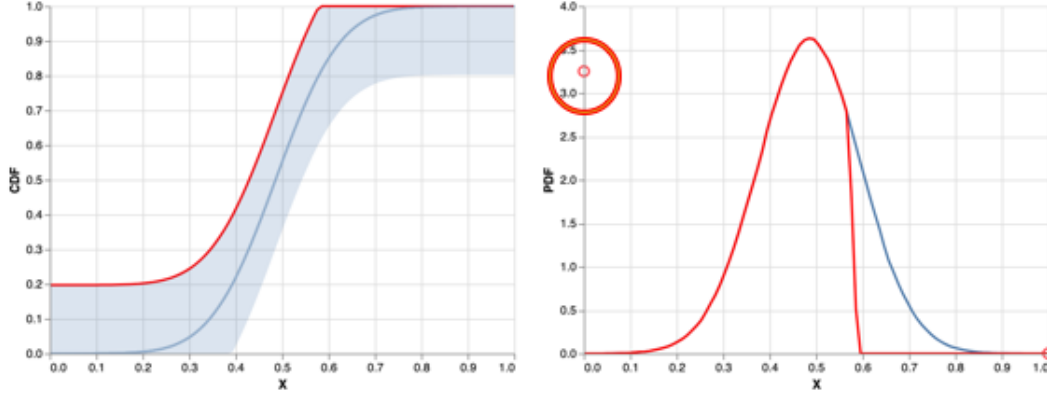


Figure 6.3: Error bounds from the DKW inequality exhibit pessimistic mass allocation.

Examples of PMA and PHOS in Error Bounders

In this section, we give examples of PMA and PHOS in the context of previously-discussed error bounders. Table 6.2 summarizes pathologies exhibited by various SSI error bounders.

Hoeffding-based. Hoeffding-based error bounders suffer from both PMA and PHOS. They have PMA since their returned CIs have widths depending only on the range of the data, $(b - a)$, and the number of samples. As such, replacing values in the sample with larger or smaller values does not affect the width of the returned error bounds. Such bounders also have PHOS since they have symmetric error, with both ends of the confidence interval depending on both range bounds a and b .

Bernstein-based. Bernstein-based error bounders do *not* suffer from PMA. To see this, notice that increasing the smallest values in some sample will also reduce the sample variance, affecting the width of the returned confidence interval, and likewise for decreasing the largest values in the sample. These bounders do, however, suffer from PHOS. Like Hoeffding-based bounders, they return confidence intervals with symmetric error, so that each end of the confidence interval is affected by both ends of the data range a and b .

Anderson/DKW-based. Anderson/DKW-based error bounders are interesting in that they suffer from PMA, but not PHOS. Consider the ε mass unaccounted for when computing a confidence lower bound using an Anderson/DKW-based bouncer. As shown in Figure 6.3, it all goes toward to lower range bound, a , which is sufficient for PMA. On the other hand, where does it come from? It comes from the ε -fraction largest observed points. This does not depend at all on the value of the upper range bound b , indicating that the confidence lower bound does not suffer from PHOS. Symmetric statements hold for the confidence upper bound, of course.

6.2.4 Problem Statement

We are now ready to give a formal problem statement.

Problem 6.1. *Design an SSI error bounder that, given a without-replacement sample from any \mathcal{D} with elements from $[a, b] \subseteq \mathbb{R}$, suffers from neither PMA nor PHOS when computing $(1 - \delta)$ error bounds for $\text{AVG}(\mathcal{D})$, for any $0 < \delta < 1$.*

Our solution to **Problem 6.1** is given in **Section 6.3** and relies on a technique we call *range trimming* in order to systematically eliminate PHOS from any range-based error bounder.

Although the solution as presented in **Section 6.3** additionally assumes knowledge of the size of \mathcal{D} , **Section 6.4** shows how our real-world implementation circumvents this limitation.

6.3 FIXING BOUNDER PATHOLOGIES

From our discussion in **Section 6.2.3**, we see that there do exist error bounders without one of either PMA or PHOS, but not without both. We first argue that error bounders without PHOS must be *asymmetric*; that is, they cannot compute bounds of the form $\hat{g} \pm \varepsilon$, where the same ε is both added and subtracted to the sample average \hat{g} in order to compute bounds. Next, we describe how to use a process we call *range trimming* to convert any symmetric, ranged-based error bounder to an asymmetric one without PHOS.

6.3.1 Decoupling Lower and Upper Bounds

Excepting an error bounder based on DKW, all of the error bounders surveyed suffer from PHOS. This is because all the other error bounders are based on concentration inequalities with *symmetric error* — that is, they return confidence intervals $[g_\ell, g_r]$ of the form $[\hat{g} - \varepsilon, \hat{g} + \varepsilon]$. At a high level, it is precisely this symmetry that causes PHOS. Although a confidence lower bound should not have any dependency on b , it is intuitively unavoidable that it has some dependency on a . Reiterating, an estimate \hat{g} could be an overestimate because of (i) not enough observed values near a , or (ii) too many observed values near b . A similar statement holds regarding confidence upper bounds, with the roles of a and b reversed.

We hypothesize that it is impossible for any confidence lower bound (resp. upper bound) to completely eliminate the dependency on a (resp. b), since it is always possible that the confidence bounding procedure got “unlucky” and operated on a sample in which values near a (resp. b) were underrepresented. Taking this hypothesis as given, this means that any symmetric confidence bounding procedure that returns bounds of the form $[\hat{g} - \varepsilon, \hat{g} + \varepsilon]$ will have ε dependent on both a and b — that is, any symmetric confidence bounding procedure will have PHOS. As such, the first

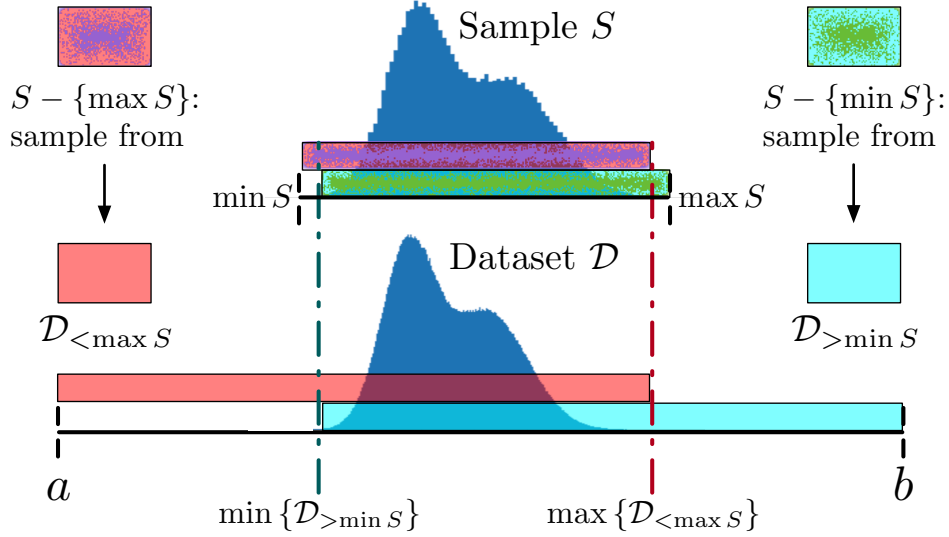


Figure 6.4: Range trimming eliminates PHOS for range-based error bounds.

step to eliminating PHOS from range-based confidence bounds is to accept asymmetric error as a hard requirement: that is, we must consider confidence bounding procedures that return bounds of the form $[\hat{g} - \varepsilon_\ell, \hat{g} + \varepsilon_r]$ for which ε_ℓ and ε_r are not necessarily equal.

6.3.2 Range Trimming

Our approach to deriving an error bound with neither PMA nor PHOS is to start with a symmetric bounder without PMA (such as that of [Algorithm 6.2](#)) and “asymmetrize” it so that Lbound becomes independent of b , and Rbound becomes independent of a , thereby eliminating PHOS. The result, given in [Algorithm 6.4](#), wraps any existing range-based error bounder.

Besides the memory required to maintain state for the left and right error bounders, S_ℓ and S_r , [Algorithm 6.4](#) requires $\mathcal{O}(1)$ extra memory to maintain the MIN and MAX element seen so far (which replace a and b when computing Rbound and Lbound, respectively).

When \mathcal{D} contains unique elements, [Algorithm 6.4](#) conceptually performs the following steps:

1. Sample S without replacement from \mathcal{D} .
2. Use Lbound to compute a $1 - \frac{\delta}{2}$ lower confidence bound for $\text{AVG}(\mathcal{D}_{<\max S})$, with $S - \{\max S\}$ as the sample, and with a and $\max S$ in place of the normal range bounds a and b , respectively.
3. Use Rbound to compute a $1 - \frac{\delta}{2}$ upper confidence bound for $\text{AVG}(\mathcal{D}_{>\min S})$, with $S - \{\min S\}$ as the sample, and with $\min S$ substituted for the range bound lower bound a .

Note that we use $\mathcal{D}_{<x}$ and $\mathcal{D}_{>x}$ as shorthand for $\mathcal{D} \cap (-\infty, x)$ and $\mathcal{D} \cap (x, \infty)$, respectively. The primary difference between these high-level steps and the pseudocode presented in [Algorithm 6.4](#) is

Algorithm 6.4: The RangeTrim meta-algorithm

Input: Dataset \mathcal{D} of N values in $[a, b]$, error prob. δ , sample size m

Output: Error bounds that fail to enclose $\text{AVG}(\mathcal{D})$ with probability $< \delta$

```
1  $S_\ell \leftarrow \text{init\_state}();$ 
2  $S_r \leftarrow \text{init\_state}();$ 
3  $a' \leftarrow \text{sample\_without\_replacement}(\mathcal{D});$ 
4  $b' \leftarrow a';$ 
5 for  $i = 1$  to  $m - 1$  do
6    $v \leftarrow \text{sample\_without\_replacement}(\mathcal{D});$ 
7    $S_\ell \leftarrow \text{update\_state}(S_\ell, \min(v, b'));$ 
8    $S_r \leftarrow \text{update\_state}(S_r, \max(v, a'));$ 
9    $a' \leftarrow \min(a', v);$ 
10   $b' \leftarrow \max(b', v);$ 
11 end
12 return  $[\text{Lbound}(S_\ell, a, b', N - 1, \frac{\delta}{2}), \text{Rbound}(S_r, a', b, N - 1, \frac{\delta}{2})];$ 
```

that [Algorithm 6.4](#) maintains min S and max S in an online, streaming fashion (so that the sample S does not need to be stored in memory), and that the confidence interval returned by [Algorithm 6.4](#) is valid even when \mathcal{D} contains duplicates (although the returned confidence bounds will bound the AVG of sets that differ slightly from $\mathcal{D}_{<\max S}$ and $\mathcal{D}_{>\min S}$). That said, we restrict our discussion and analysis to the case where \mathcal{D} contains unique elements, for simplicity.

Correctness of [Algorithm 6.4](#) crucially depends on the fact that, conditioned on the value of max S (and for any such value), the remaining elements in S (namely $S - \{\max S\}$) constitute a uniform without-replacement sample from $\mathcal{D}_{<\max S}$, with a symmetric statement for min S and $S - \{\min S\}$. At a high level, this means that a confidence lower bound computed over $S - \{\max S\}$ is a valid confidence lower bound for $\text{AVG}(\mathcal{D}_{<\max S})$, and since $\text{AVG}(\mathcal{D}_{<\max S}) \leq \text{AVG}(\mathcal{D})$, it is also a valid confidence lower bound for $\text{AVG}(\mathcal{D})$, with symmetric statements holding for the confidence upper bound, $S - \{\min S\}$, and $\mathcal{D}_{>\min S}$. These core ideas are illustrated in [Figure 6.4](#).

Tradeoffs. When either of the range bounds a or b are actually observed in a sample, [Algorithm 6.4](#) will compute CIs that are looser than necessary. For example, suppose that, for $[a, b] = [0, 1]$, a sample with a single 0 is observed. When computing the upper confidence bound, [Algorithm 6.4](#) sets $a' = 0$, which is the same as the original a ; however, it also throws away this 0 point which would have pulled the upper confidence bound lower. The upper confidence bound actually computed will end up being larger than necessary, corresponding to an unnecessarily loose interval. Fortunately, we have found that, in practice, losing this single sample does not significantly increase the number of samples needed to achieve some desired CI width.

6.3.3 Correctness of Algorithm 6.4

We now give a theorem on the correctness of Algorithm 6.4.

Theorem 6.2. *Given SSI range-based bounders Lbound and Rbound for computing lower (resp. upper) confidence bounds and a dataset \mathcal{D} of N unique values known to all fall in the interval $[a, b] \subseteq \mathbb{R}$, Algorithm 6.4 returns a $(1 - \delta)$ confidence interval for $\text{AVG}(\mathcal{D})$.*

Please see the appendix (§A.6) for the proof.

6.4 SYSTEM CONSIDERATIONS

In this section, we address a number of implementation issues that become pertinent when applying techniques of previous sections in a real system. Although the techniques presented in this section are auxiliary to our primary contribution and can be used with any CI approach, they are developed with SSI error bounders and strong probabilistic guarantees in mind. First, we describe how to augment the techniques of Section 6.3, which apply for a fixed sample size taken without replacement from a finite dataset of known size, with locality-aware *scan-based* without-replacement sampling that need not know N , and we further describe how to use this layout to facilitate SUM and COUNT aggregations (§6.4.1). Next, we describe an *optional stopping* routine that does not require a sample size to be specified up-front (§6.4.2). Finally, we describe an *active scanning* architectural optimization that prioritizes samples that facilitate early termination (§6.4.3), all without losing guarantees proved in Section 6.3.

These system details are implemented within the context of FASTFRAME, which is relational column store for approximate report generation with guarantees that generalizes that FASTMATCH architecture from Chapter 5. FASTFRAME uses the error bounders from Section 6.3 and pairs them with a practical architecture for without-replacement sampling. As in Chapter 5, FASTFRAME uses block-based bitmaps over categorical attributes for efficient processing of queries with predicates or groups. Furthermore, for continuous attributes, FASTFRAME stores the minimum and maximum values in a catalog, to be used as the range bounds a and b for the desired range-based error bounder.

6.4.1 Scan-Based Sampling for DB Aggregates

We now describe how FASTFRAME implements without-replacement sampling in a locality-aware manner by *scanning* over pre-shuffled data, and furthermore how this approach can be used to compute CIs for COUNT and SUM. As in Chapter 5, the up-front shuffling cost need only be paid once in order to facilitate many queries, although care must be taken to set the error probability

δ small enough when running multiple queries to avoid losing error bounder guarantees. This approach is not new and has been used in prior work [157, 158, 159, 178]. We begin by introducing *scrambles* and *aggregate views*:

Definition 6.4 [Scramble]. *A scramble is an ordered copy of a relational table that has been permuted randomly, allowing for scan-based without-replacement sampling.*

Note that there exist external shuffling techniques that can handle data too large to fit in memory [179].

Scanning a continuous column in a scramble is equivalent to sampling without replacement. In fact, scanning any subset of data in a continuous column in a scramble (assuming the subset is chosen without knowledge of the order of data) is also equivalent to sampling without replacement, so that scanning a scramble can be used to sample without replacement for any aggregate appearing in a query containing arbitrary filters or GROUP BY clauses. We call such subsets *aggregate views*:

Definition 6.5 [Aggregate View]. *An aggregate view for some aggregate A appearing in a query (possibly belonging to a group induced by a GROUP BY clause) is the set of values in a scramble that contribute toward the computation of A .*

Choosing δ to Maintain Guarantees. Note that δ must be divided by the number of aggregate views in a query (or an upper bound) to preserve error guarantees (via a union bound). Furthermore, to guarantee (via another union bound) that the probability of one or more queries in a workload giving incorrect results is at most δ , we must further divide by (an upper bound on) the number of queries in the workload before supplying it as a parameter to an error bounder. For this reason, we choose $\delta = 10^{-15}$ when we discuss our empirical study in Section 6.5, since even for large workloads comprised of, say, 10^7 queries, the probability of one or more mistakes will still be at most 10^{-8} . Indeed, our techniques are specifically targeted toward the case wherein it is known ahead of time that a large (but possibly unknown) number of exploratory queries will be issued, so that the cost to materialize the scramble is justified. Note that, provided δ is properly decayed, guarantees hold when the same scramble is used across an entire query workload; i.e., it is not necessary to reshuffle the scramble between queries.

Computing CIs for COUNT. Ensuring that data in a scramble are permuted randomly makes it easy to compute bounds on the selectivities of aggregate views, and by extension on the COUNT of tuples in each aggregate view, using existing techniques [180, 181]. To outline the basic idea, consider that one can conceptually assign each row of a scramble a 1 if it belongs to the aggregate view of interest, and a 0 otherwise. The AVG of this “derived” view (over the whole scramble) is exactly the selectivity of the aggregate view, and we can use a Hoeffding-Serfling-based bounder to compute a CI for the selectivity (using range bounds of $a = 0$ and $b = 1$). Multiplying these

bounds by the total number of rows in the scramble then yields a CI for the COUNT of rows that participate in the aggregate view.

In more detail, for a scramble with R rows, N of which are in the sample view V for a query Q , the number of rows seen that belong to V , m_v , after scanning r rows of the scramble is a hypergeometric random variable [143] whose mean is the selectivity σ_v of V multiplied by r , $\sigma_v \cdot r$. One could use bounds specifically tailored to the hypergeometric distribution (or even perform an exact computation) to compute an upper bound on σ_v that holds w.h.p., but in this work we use a simple strategy that uses Hoeffding-Serfling to bound σ_v , stated as follows.

Lemma 6.4. *The probability that a scan of a scramble of size R that has processed r rows so far sees fewer than $(\sigma_v - \varepsilon) \cdot r$ or more than $(\sigma_v + \varepsilon) \cdot r$ rows belonging to V is at most δ , for $\varepsilon = \sqrt{\frac{\log(2/\delta)}{2r} \cdot (1 - \frac{r-1}{R})}$.*

Proof. Follows immediately from application of the Hoeffding-Serfling inequality [163].

Lemma 6.4 implies that, for a scan that has seen m_v rows so far belonging to V , σ_v is within

$$\hat{\sigma}_v \pm \varepsilon = \frac{m_v}{r} \pm \sqrt{\frac{\log(2/\delta)}{2r} \cdot \left(1 - \frac{r-1}{R}\right)} \quad (6.11)$$

w.h.p. This in turn implies that N , the number of tuples belonging to V , is within $[N^-, N^+] = [(\hat{\sigma}_v - \varepsilon) \cdot R, (\hat{\sigma}_v + \varepsilon) \cdot R]$, w.h.p.

Combining Lemma 6.4 with error bounders to compute CIs for AVG with unknown COUNT.

For error bounders Lbound and Rbound of the form described in Section 6.3 that require the data range bounds a and b as well as the data size N , a $(1 - \delta)$ confidence interval is computed as

$$[\text{Lbound}(S, a, b, N, \delta/2), \text{Rbound}(S, a, b, N, \delta/2)] \quad (6.12)$$

The following theorem describes how to use Lemma 6.4 to compute $(1 - \delta)$ error bounds when N is unknown.

Theorem 6.3. *Consider a query Q operating over some size- R scramble with corresponding sample view V . Suppose a scan the scramble that has processed k rows so far has seen m_v rows belonging to V (from which S is computed). Letting*

$$N^+ = \left(\frac{m_v}{r} + \sqrt{\frac{\log(1/(1 - \alpha) \cdot \delta)}{2r} \cdot \left(1 - \frac{r-1}{R}\right)} \right) \cdot R \quad (6.13)$$

then the interval

$$[\text{Lbound}(S, a, b, N^+, \alpha \cdot \delta/2), \text{Rbound}(S, a, b, N^+, \alpha \cdot \delta/2)] \quad (6.14)$$

is a $(1 - \delta)$ confidence interval for the mean of V , for any $\alpha \in (0, 1)$.

Proof. Conditioning over whether $N \leq N^+$ or $N > N^+$, the probability that the aforementioned interval $[L, R]$ fails to contain the desired mean μ is

$$\mathbb{P}(\mu \notin [L, R] \mid N > N^+) \cdot \mathbb{P}(N > N^+) \quad (6.15)$$

$$+ \mathbb{P}(\mu \notin [L, R] \mid N \leq N^+) \cdot \mathbb{P}(N \leq N^+) \quad (6.16)$$

$$\leq \mathbb{P}(N > N^+) + \mathbb{P}(\mu \notin [L, R] \mid N \leq N^+) \quad (6.17)$$

By [Lemma 6.4](#), the first probability is at most $(1 - \alpha) \cdot \delta$, and Lbound and Rbound are such that the probability of the second term is at most $\alpha \cdot \delta$. The sum of these equals δ , implying that the interval is a valid $(1 - \delta)$ confidence interval, completing the proof.

Throughout [Section 6.5](#), we fix $\alpha = 0.99$, giving most of the weight to the confidence interval computation, corresponding to a looser upper bound for N .

Computing CIs for SUM. Now that we have established how to compute CIs for AVG and COUNT, we briefly describe how to combine these two techniques to compute CIs for SUM. Given a $(1 - \frac{\delta}{2})$ confidence interval for COUNT as $[c_\ell, c_r]$ and a $(1 - \frac{\delta}{2})$ confidence interval for SUM as $[g_\ell, g_r]$, union bounding gives $[c_\ell \cdot g_\ell, c_r \cdot g_r]$ as a $(1 - \delta)$ confidence interval for SUM.

Scramble Use and Maintenance under Updates. There exist a few ways to leverage the scramble for analytical queries even for hybrid workloads that additionally involve deletes, updates, and insertions. First, deletes can be handled by simply writing a tombstone to any deleted tuples, and (non-insertion) updates can be handled by simply modifying the updated tuple in-place. In this case, processing the tuples sequentially while ignoring tombstones will remain equivalent to sampling without replacement.

For insertions, there are two options: the first is to leave some holes in the scramble to allow for insertions in random positions, at the cost of increasing space and query time. The second option is to write all updates to a separate smaller “insertion table” that is scanned in full for every query, and then combined with the results of the scramble in a manner which preserves guarantees. This insertion table can then be periodically merged with the scramble and reshuffled. Overall, with some minor extensions inspired from existing big data systems (such as Bigtable [\[182\]](#)) that handle updates by keeping them separate and periodically merging them, scrambles can be readily retrofitted to handle non-read-only workloads.

Algorithm 6.5: The OptStop meta-algorithm

Input: Dataset \mathcal{D} of N values in $[a, b]$, error probability δ

Output: Error bounds that fail to enclose $\text{AVG}(\mathcal{D})$ with probability $< \delta$

```
1  $S \leftarrow \text{init\_state}();$ 
2 for  $k = 1$  to  $\infty$  do
3   for  $i = 1$  to  $B$  do
4      $v \leftarrow \text{sample\_without\_replacement}(\mathcal{D});$ 
5      $S \leftarrow \text{update\_state}(S, v);$ 
6   end
7    $\delta' \leftarrow (6/\pi^2) \cdot (\delta/k^2);$ 
8    $L_k \leftarrow \text{Lbound}(S, a, b, \frac{\delta'}{2});$ 
9    $R_k \leftarrow \text{Rbound}(S, a, b, \frac{\delta'}{2});$ 
10  if  $\text{should\_stop}([\max_{j \leq k} \{L_j\}, \min_{j \leq k} \{R_j\}])$  then
11    break;
12  end
13 end
14 return  $[\max_k \{L_k\}, \min_k \{R_k\}]$ 
```

6.4.2 Optional Stopping

The techniques discussed in [Section 6.3](#) describe how to compute high-probability bounds on error given statistics computed from a particular sample of m datapoints. Fixing a sample size ahead of time is oftentimes impractical, since it is usually unknown how many samples are needed to ensure CIs that are “just tight enough” to facilitate downstream applications on the part of the user or the system. For example, one approach (e.g. taken by VerdictDB [\[77\]](#)) is to first compute error bounds around an approximate aggregate, and then run an exact query if these bounds are too loose.

Another approach, which we take in this chapter, is to continue taking samples until a bound on the error is provably small enough. For this approach, care must be taken to avoid losing guarantees offered by range-based error bounders, since the tighter of two $(1 - \delta)$ confidence intervals for a particular aggregate is itself not necessarily a $(1 - \delta)$ confidence interval.

Various techniques have been developed for computing *sequentially-valid* confidence intervals as new samples are taken [\[6, 82, 183, 184, 185, 186\]](#). In addition to techniques for sequential estimation [\[183\]](#) for sequences of i.i.d. random variables from a known family of distributions, various concentration results applicable to AVG which make no distributional assumptions have likewise been derived [\[6, 184\]](#). Unfortunately, these existing results are derived from variants of Hoeffding’s inequality, and therefore suffer from PMA. For the sake of simplicity, we use a much simpler meta-algorithm that can be used in conjunction with any range-based error bouncer, including

those that leverage our RangeTrim technique, given in [Algorithm 6.5](#). Although [Algorithm 6.5](#) requires more samples than the aforementioned techniques when used in conjunction with Hoeffding- or Hoeffding-Serfling-based error bounders, we consider the tradeoff worthwhile due to its generality and simplicity and leave better sequential error bounders to future work.

Analysis of [Algorithm 6.5](#). [Algorithm 6.5](#) proceeds in “rounds”, with each iteration of the outer loop on [line 2](#) forming a round. During each round, B without-replacement samples are taken and used to incrementally update the state of any range-based error bouncer that implements our interface from [Section 6.2.2](#). At the end of each round, confidence intervals are recomputed, with the input error probability δ' decayed “enough” to ensure that the probability of error across *all* rounds is at most δ . If the stopping condition on [line 10](#) is met, then the algorithm terminates; otherwise, it proceeds to the next round, decaying δ' appropriately to control the overall error probability.

We now give a proof of correctness of [Algorithm 6.5](#).

Theorem 6.4. *With probability at least $(1 - \delta)$, the $\{L_k\}$ and $\{R_k\}$ computed by [Algorithm 6.5](#) satisfy $AVG(\mathcal{D}) \in [L_k, R_k]$ for every k in the outer loop. In particular, $AVG(\mathcal{D}) \in [\max_{k \geq 1} L_k, \min_{k \geq 1} R_k]$ with probability at least $(1 - \delta)$.*

Proof. Denote the δ' used at iteration k with δ_k . Union bounding over rounds, the probability of a mistake is at most

$$\sum_{k \geq 1} \delta_k = \frac{6}{\pi^2} \sum_{k \geq 1} \frac{\delta}{k^2} = \frac{6}{\pi^2} \cdot \frac{\pi^2}{6} \delta = \delta \quad (6.18)$$

using the identity $\sum_{k \geq 1} \frac{1}{k^2} = \frac{\pi^2}{6}$, completing the proof.

FASTFRAME performs I/O at the level of blocks, so instead of computing bounds every B samples as described in the pseudocode of [Algorithm 6.5](#), we compute bounds after every B block read. In our experiments ([Section 6.5](#)), we set $B = 40000$. We leave development of alternative approaches to future work.

Stopping Conditions for [Algorithm 6.5](#). Correctness of [Algorithm 6.5](#) is independent of whether the error bouncer uses our RangeTrim technique, and it is furthermore independent of stopping condition. We consider several stopping conditions used in our system implementation:

- ❶ **Desired Samples Taken** ($c \geq m$): If a fixed number of samples are requested, do not use [Algorithm 6.5](#); instead, terminate query processing once a desired number of tuples contribute to the partial aggregate(s) in the query.
- ❷ **Sufficient Absolute Accuracy** ($\hat{g}_r - \hat{g}_\ell < \varepsilon$): The interval width is sufficiently small.
- ❸ **Sufficient Relative Accuracy** ($\max\{\frac{g_r - \hat{g}}{g_r}, \frac{\hat{g} - g_\ell}{g_\ell}\} < \varepsilon$): The interval width is sufficiently small (relative to the possible correct values implied by the interval).

- ④ **Threshold Side Determined** ($v \notin [g_\ell, g_r]$): The interval does not contain some threshold value v , indicating that the true AVG is w.h.p. either less than or greater than the threshold v .
- ⑤ **Top- or Bottom- K Separated**: In a query with multiple groups, the error bounds of the groups with either K smallest or largest aggregates do not intersect those of any of the remaining groups.
- ⑥ **Groups Ordered Correctly**: In a query with multiple groups, the error bounds for each group intersect none of the other groups' error bounds, indicating that the correct ordering of group aggregates has been determined [6].

Different stopping conditions apply to different queries. For example, stopping conditions ③ and ④ might be used for the query in Figure 6.2.

6.4.3 Active Scanning

For queries with GROUP BYs, different groups may require different numbers of samples to achieve stopping conditions of the types considered in Section 6.4.2. For simple scans that simply read blocks of the scramble in the order in which they appear, it is impossible to control the relative number of tuples for each group, leading to potential inefficiencies. For example, consider one of the queries in our experiments, F-q2, which selects airlines with average delay above some threshold. This query uses stopping condition ④ in order to determine when to terminate, since, when this stopping condition has been achieved, it has been determined w.h.p. whether each airline has average delay above or below the threshold. Those groups (airlines) for which the average delay is near \$thresh require more samples than those for which the average delay is far from \$thresh in order to achieve condition ④. If these groups are sparse within the scramble, a scan will look at much more data than necessary.

For this reason, we process queries that perform GROUP BYs with an adaptive sampling approach using *active scanning*. Active scanning uses block-based bitmap indexes to efficiently check whether a block contains tuples for any active group — such groups are marked for processing, and blocks without tuples for any active group are skipped, since they are unlikely to help achieve early termination. The notion of an active group depends on the stopping condition, but in brief, active groups are groups that should be prioritized.

Active Groups for Stopping Conditions. We now describe how we determine active groups, or groups that should be prioritized for sampling, for each of the stopping conditions discussed in §6.4.2.

- ① (Desired Samples Taken): Under this condition, we consider a group active as long as fewer than the desired m samples have been taken that contribute to that group's aggregate.
- ② (Sufficient Absolute Accuracy): We consider a group active as long as its confidence bounds

Dataset	Size	Tuples	Columns	Replications
FLIGHTS	32 GB	606 mil.	5	5×
TAXI	36 GB	679 mil.	4	4×
POLICE	12 GB	292 mil.	3	72×

Table 6.3: Dataset descriptions.

exceed ε in width; i.e., $\hat{g}_r - \hat{g}_\ell \geq \varepsilon$.

③ (Sufficient Relative Accuracy): Same as the previous, but a group is active if $\max\{\frac{g_r - \hat{g}}{g_r}, \frac{\hat{g} - g_\ell}{g_\ell}\} \geq \varepsilon$.

④ (Threshold Side Determined): A group is active as long as the threshold side has not been determined; i.e., $v \in [g_\ell, g_r]$.

⑤ (Top- or Bottom- K Separated): Consider sorting the aggregates for all the groups in increasing order. A group among the top- K is active if its lower confidence bound g_ℓ crosses the midpoint between the aggregate value for the smallest of the top- K and the largest of the bottom $N - K$. Likewise, a group among the bottom $N - K$ is active if its upper confidence bound g_r crosses this midpoint. Analogous statements hold if we consider the separation between the bottom- K and the top $N - K$ as the stopping condition, but with the aggregates sorted in decreasing order.

⑥ (Groups Ordered Correctly): A group is active if its interval $[g_\ell, g_r]$ intersects the interval of any other group.

Lookahead. We furthermore accelerate active scanning with a lookahead technique from [Chapter 5](#), which we now briefly review. Instead of checking each block one by one for whether it contains tuples for any active group, recall that active scanning *with lookahead* checks, for each active group, whether each block in a batch of 1024 blocks contains any tuples for that group. By iterating over an entire batch of 1024 blocks for a given active group, bitmaps for the group tend to be in cache more often, making the index lookup operation more efficient. Please refer to [Section 5.4.2](#) for more details. In our experiments in [Section 6.5](#), we set the block size to 64 rows, so a batch of 1024 blocks contains a total of 65536 rows.

6.5 EMPIRICAL STUDY

In this section, we perform an extensive empirical evaluation of various error bounders and sampling strategies on real data.

Query	Stop When	Parameters Varied
F-q1	(③) $\max\{\frac{g_r - \hat{g}}{g_r}, \frac{\hat{g} - g_\ell}{g_\ell}\} < \varepsilon$	\$airport (Figure 6.7), ε (Figure 6.8(a))
F-q2	(④) $\$thresh \notin [g_\ell, g_r]$	$\$thresh$ (Figure 6.8(b))
F-q3	(⑤) bottom-2 separated	$\$min_dep_time$ (Figure 6.9)
F-q4	(④) $10 \notin [g_\ell, g_r]$	N/A
F-q5	(④) $0 \notin [g_\ell, g_r]$	N/A
F-q6	(⑤) top-5 separated	N/A
F-q7	(⑥) groups ordered	N/A
F-q8	(⑤) top-1 separated	N/A
F-q9	(⑤) top-1 separated	N/A
T-q1	(⑤) top-1 separated	N/A
T-q2	(⑤) top-1 separated	N/A
P-q1	(⑤) top-1 separated	N/A
P-q2	(⑤) top-1 separated	N/A

Table 6.4: Summary of stopping conditions and template parameters used for queries provided in Figures 6.5 and 6.6. Template variable arguments shown in blue.

6.5.1 Datasets and Queries

We evaluate various error bounding techniques on publicly available FLIGHTS, TAXI, and POLICE datasets [134, 161, 162]. For FLIGHTS, we extract five attributes corresponding to the origin airport, airline, departure delay, departure time, and day of week. To ensure sufficient scale of the data, we perform 5 replications, giving a 32 GiB dataset of 606 million tuples in total. For TAXI, we extract four attributes (for hour of day, passenger count, trip distance, and fare amount) and perform 4 replications. For POLICE, we extract 3 attributes (for number of violations, officer race, and driver age) and perform 72 replications. These datasets are summarized in Table 6.3. We eliminated rows with “N/A” or erroneous values for any column appearing in one or more of our queries.

Queries and Query Templates. We evaluate our techniques on a diverse set of queries that include various filters and GROUP BY clauses and exercise all the stopping conditions described in Section 6.4.2 (except conditions ① and ②, which gives similar behavior to condition ③). The queries themselves are given in Figures 6.5 and 6.6, and the accompanying stopping conditions are summarized in Table 6.4. Additionally, several queries are parametrized, in order to reveal interesting data-dependent behavior by varying corresponding parameters. Any query parameters varied are also summarized in Table 6.4, with parameters shown in blue.

```

# F-q1: avg delay for $airport
SELECT AVG(DepDelay) FROM flights WHERE Origin = $airport

# F-q2: airlines with avg delay above $thresh
SELECT Airline FROM flights
GROUP BY Airline HAVING AVG(DepDelay) > $thresh

# F-q3: 2 airlines with min avg delay after $min_dep_time
SELECT Airline FROM flights WHERE DepTime > $min_dep_time
GROUP BY Airline ORDER BY AVG(DepDelay) ASC LIMIT 2

# F-q4: whether ORD has avg delay > 10
SELECT (CASE WHEN AVG(DepDelay) > 10 THEN 1 ELSE 0 END)
FROM flights WHERE Origin = 'ORD'

# F-q5: airports with negative avg departure delay
SELECT Origin FROM flights
GROUP BY Origin HAVING AVG(DepDelay) < 0

# F-q6: 5 worst days for afternoon delays across airports
SELECT DayOfWeek, Origin FROM flights
WHERE DepTime > 1:50pm GROUP BY DayOfWeek, Origin
ORDER BY AVG(DepDelay) DESC LIMIT 5

# F-q7: avg delay by day of week for airline HP
SELECT DayOfWeek, AVG(DepDelay) FROM flights
WHERE Airline = 'HP' GROUP BY DayOfWeek

```

Figure 6.5: SQL for queries (part 1). Template parameters are shown in \$blue.

6.5.2 Experimental Setup

The core of our experiments consists of two ablation studies, intended to evaluate the impact of both our error bounder innovations *and* that of our architectural innovations. In particular, we evaluate various error bounders with and without our RangeTrim technique developed in [Section 6.3](#), and for the best error bounder (Bernstein+RT), we furthermore evaluate the impact of leaving out features of our active scanning sampling strategy described in [Section 6.4](#).

We set $\delta = 10^{-15}$ as the default for all queries unless otherwise noted, as we expect users of with-guarantees AQP to desire results that are correct in an *effectively deterministic* manner.

Approaches. We used the following strategies to bound error when running queries in [Figures 6.5](#) and [6.6](#):

Error Bounders.

- Bernstein+RT. This uses the empirical Bernstein-Serfling error bounder described in [Section 6.2.3](#), coupled with our RangeTrim technique described in [Section 6.3](#), which eliminates PHOS.


```
# F-q8: origin airport with highest departure delay
SELECT Origin FROM flights GROUP BY Origin
ORDER BY AVG(DepDelay) DESC LIMIT 1
```

```
# F-q9: airline with maximum avg delay
SELECT Airline FROM flights GROUP BY Airline
ORDER BY AVG(DepDelay) DESC LIMIT 1
```

```
# T-q1: Hour with maximum avg distance for trips with 4 passengers
SELECT HourOfDay FROM taxi WHERE passenger_count=4
GROUP BY HourOfDay ORDER BY AVG(trip_distance) DESC LIMIT 1
```

```
# T-q2: Hour with maximum avg fare for trips with 3 passengers
SELECT HourOfDay FROM taxi WHERE passenger_count=3
GROUP BY HourOfDay ORDER BY AVG(fare_amount) DESC LIMIT 1
```

```
# P-q1: Officer ethnicity associated with maximum average driver violations per stop
SELECT OfficerRace FROM police
GROUP BY OfficerRace ORDER BY AVG(violations) DESC LIMIT 1
```

```
# P-q2: Driver age associated with maximum average driver violations per stop
SELECT DriverAge FROM police
GROUP BY DriverAge ORDER BY AVG(violations) DESC LIMIT 1
```

Figure 6.6: SQL for queries (part 2). Template parameters are shown in \$blue.

- Bernstein. Same as the previous, but without RangeTrim. Bernstein and Bernstein+RT are included to evaluate the impact of an error bounder without PMA.
- Hoeffding+RT. This uses the Hoeffding-Serfling error bounder described in Section 6.2.3, coupled with our RangeTrim technique described in Section 6.3, which eliminates PHOS from Hoeffding (but does not fix PMA).
- Hoeffding. Same as the previous, but without RangeTrim.
- Exact. This strawman approach eschews approximation and runs queries exactly, to serve as a simple baseline.

We furthermore used the following strategies for sampling when running queries in Figures 6.5 and 6.6:

Sampling Strategies.

- ActivePeek. This uses the active scanning technique to prioritize groups that are preventing satisfaction of various stopping conditions, along with cache-efficient queries to bitmaps with lookahead (see Section 6.4.3 for details).
- ActiveSync. This uses active scanning, but processes each block synchronously when deciding whether to read it, incurring high overhead since queries to bitmaps typically result in cache misses.

- **Scan.** This strategy does not leverage bitmaps in order to decide whether to read a block for active scanning (but may leverage bitmaps for evaluation of whether a block contains tuples that satisfy a fixed predicate, such as the one appearing in **F-q1**). Without any predicate, this approach simply processes all blocks in the scramble sequentially. Note that the Exact baseline described previously always uses Scan, as only approximate approaches can prune groups.

Environment. Experiments were run on single Intel Xeon E5-2630 node with 125 GiB of RAM and with 8 physical cores (16 logical) each running at 2.40 GHz, although we restrict our experiments to a single thread, noting that our techniques can be easily parallelized. The Level 1, Level 2, and Level 3 CPU cache sizes are, respectively: 512 KiB, 2048 KiB, and 20480 KiB. We ran Linux with kernel version 2.6.32. We report results for data stored in-memory, since the cost of main memory has decreased to the point that many interactive workloads can be performed entirely in-core. Each approximate query was started from a random position in the shuffled data. We found wall clock time to be stable for all approaches, and report times as the average of 3 runs for all methods.

6.5.3 Metrics

We gather several metrics in order to test two hypothesis: one, that our error bounding strategies in conjunction with our sampling strategies lead to speedups over simpler baselines; and two, that they do so without sacrificing correctness of query results.

Correctness of Query Results. The most important metric is the fraction of queries run that returned correct results, which we discuss briefly here. *Across all methods, all queries, and all parameter settings, results either matched the ground truth determined from an Exact evaluation, or were within error tolerance in the case of **F-q1** and **F-q7**.* This is expected, given that we consider SSI error bounders with strong probabilistic guarantees, coupled with the fact that our RangeTrim technique and system architecture do not compromise these guarantees. As such, we expect fewer than $\delta = 10^{-15}$ fraction of queries to yield incorrect results, which rounds down to 0.

For the remaining experiments, we focus on the following metrics:

Estimate Error. For a given requested error bound ε supplied to applicable queries, we measure the actual error. The observed error should always fall within the requested error bound.

Wall-Clock Time. Our primary metric evaluates the end-to-end time required for various error bounders and various sampling strategies (where the Exact baseline is included as a “sampling strategy”), across all the queries considered.

Number of Blocks Fetched. We also measure the number of blocks fetched from main memory into CPU cache when using various approaches. This is mainly due to the fact that error bounders incur additional CPU overhead and therefore wall-clock time, with Bernstein and Bernstein+RT

Query		Avg Speedup over Exact (raw time in (s))			
	Exact (s)	Hoeffding	Hoeffding+RT	Bernstein	Bernstein+RT
F-q1['ORD',0.5]	20.7	58.0× (0.4)	59.6× (0.3)	2129× (0.01)	2160 × (0.01)
F-q2[\$thresh=0]	42.2	233× (0.2)	316× (0.1)	2405× (0.02)	4683 × (0.01)
F-q3[10:50pm]	25.3	1.1× (23.4)	1.7× (15.1)	6.5× (3.9)	13.8 × (1.8)
F-q4	20.7	12.8× (1.6)	12.7× (1.6)	922× (0.02)	925 × (0.02)
F-q5	47.7	0.7× (68.8)	1.1× (44.4)	2.2× (21.4)	4.2 × (11.5)
F-q6	66.4	1.1× (62.4)	1.1× (58.7)	11.2× (6.0)	16.7 × (4.0)
F-q7	29.8	1.0× (30.0)	1.0× (29.1)	2.6× (11.6)	2.9 × (10.4)
F-q8	47.7	2.0× (23.3)	1.0× (47.6)	9.1× (5.3)	9.5 × (5.0)
F-q9	38.2	0.9× (41.0)	1.0× (38.6)	123× (0.3)	130 × (0.3)
T-q1	32.4	2.2× (14.6)	3.4× (9.5)	17.8× (1.8)	29.4 × (1.1)
T-q2	39.6	1.6× (24.5)	2.1× (19.0)	19.3× (2.0)	27.0 × (1.5)
P-q1	20.9	17.8× (1.2)	17.9× (1.2)	311× (0.07)	314 × (0.07)
P-q2	22.5	11.7× (1.9)	11.3× (2.0)	18.9× (1.2)	20.8 × (1.1)

Table 6.5: Summary of average query speedups and latencies for various error bounders.

incurring the highest overhead, so measuring blocks fetched for these approaches removes this confounding variable by decoupling performance from CPU attributes.

6.5.4 Results

In this section, we present results of our empirical study.

Impact of Error Bounder Used

Summary. Using the Bernstein+RT error bounder resulted in typical speedups of at least 10× over Exact (for 10 out of 13 queries) and Hoeffding (for 9 out of 13 queries), and additionally was almost always on par or better than Bernstein (up to 2× faster).

We evaluate Hoeffding+RT and Bernstein+RT error bounders, along with Hoeffding and Bernstein (to ablate our RangeTrim technique) and an Exact query processor (to ablate any benefits due to approximation) against all the queries in Figures 6.5 and 6.6, with the resulting time measurements summarized in Table 6.5.

First we note that all error bounders incur additional overhead — in the case of F-q5 where techniques like Hoeffding and Hoeffding+RT needed to process all the data in order to terminate (due to PMA), they actually ran *more slowly* than Exact. Using Bernstein, which does not suffer from PMA, yielded significant benefits over Exact, Hoeffding, and Hoeffding+RT across all queries. In

	Avg Speedup over Scan (time in seconds)		
	Scan	ActiveSync	ActivePeek
F-q5	40.1	2.0× (19.6)	3.5× (11.4)
F-q6	4.2	1.1× (3.8)	1.1× (3.9)
F-q7	10.3	1.0× (10.1)	1.0× (10.4)
F-q8	40.4	3.2× (12.7)	8.2× (5.0)
T-q1	1.2	1.2× (1.0)	1.1× (1.1)
T-q2	1.6	1.2× (1.4)	1.1× (1.5)
P-q2	11.3	7.1× (1.6)	10.4× (1.1)

Table 6.6: Summary of query speedups and latencies for various sampling strategies, restricted to GROUP BY queries that take more than 500ms for Scan with Bernstein+RT.

cases where Hoeffding and Hoeffding+RT showed improvements over Exact, Bernstein amplified these improvements (**F-q1**, **F-q2**).

Using RangeTrim in conjunction with both Hoeffding and Bernstein typically led to similar performance, with a few queries exhibiting clearly superior performance (**F-q5**, **F-q6**, and **F-q3**). These queries have the following in common: they all have sparse groups with low selectivity (either because of the large number of groups in the case of **F-q5** and **F-q3**, or because of the restrictive filter in the case of **F-q5**), and they are all “easy” to approximate, in that none of the groups require too many samples in order to achieve the relevant stopping condition. (**F-q8** also has many groups, but some of them require many samples due to a large number of airports with average delay near the max.) This is an ideal condition for Bernstein+RT to show benefit: sparse groups will bottleneck the query, but RangeTrim will achieve termination faster since these sparse groups tend to have fewer outliers than do non-sparse groups. For such bottlenecking sparse groups, the range bounds for the DepDelay column are overly-conservative and dominate the sampling complexity. In this case, Bernstein, which has PHOS, will require twice as many samples for such groups — and since these groups are the bottleneck, it will require roughly twice as much time, an intuition reflected in **Table 6.5**.

Impact of Sampling Strategy Used

Summary. ActiveSync sampling was typically on par or better than Scan, and ActivePeek sampling was typically on par or better than ActiveSync. ActiveSync significantly outperformed Scan on **F-q5**, **F-q8**, and **P-q1** (by more than 3×), and ActivePeek significantly outperformed ActiveSync on the same set of queries.

We evaluate the impact of various sampling strategies when used in conjunction with the Bernstein+RT error bounder, the results of which are summarized in Table 6.6. In some cases (F-q5 and F-q8), the performance of the Scan baseline when used in conjunction with Bernstein+RT was on par with that of the Exact baseline, indicating that some form of block skipping can be crucial for queries with GROUP BYs. When implementing active scanning block by block as in ActiveSync, the improvement was most significant for F-q5, F-q8, and P-q1, with ActivePeek showing even further improvements for these queries. It is no coincidence that these are the same queries for which Scan performance using approximation is similar to Exact performance. This indicates that there were a few sparse groups preventing termination when Scan is used, which is the very case for which the greatest benefit can be derived from (an efficient implementation of) block skipping.

Impact of Data and Query Characteristics

To better understand various data- and query-dependent aspects of our techniques, we now study the effect of varying the parameters supplied to F-q1, F-q2, and F-q3.

Selectivity σ of filter.

Summary. As the fraction of tuples passing F-q1's filter increases, wall clock time and blocks fetched both increase rapidly, then decrease, with RangeTrim giving the most benefit in the case of predicates with intermediate selectivity.

Different Origin attribute values used for filtering F-q1 have different selectivities. By varying the filter attribute value, we reveal interesting behavior impacted by the selectivity of the filter. (We consider selectivity as a number and not a quality, so that larger proportions of tuples satisfy predicates with higher selectivity.) For all four error bounding techniques considered, wall time and blocks fetched are plotted versus query selectivity in Figure 6.7. Bernstein and Bernstein+RT are plotted separately from Hoeffding and Hoeffding+RT for presentation.

For Hoeffding and Hoeffding+RT bounders, as selectivity increases, both wall-clock time and number of blocks fetched first increase rapidly, then decrease, in a strongly correlated fashion. This is likely because the sparsest filters require examining all the data before terminating, obviating early stopping benefits. After a certain point, however, early termination kicks in, happening more quickly as fewer tuples are filtered. The selectivity threshold for early termination appears to be much lower for Bernstein and Bernstein+RT bounders, which explains why wall-clock time and number of blocks fetched appear to be strictly decreasing with selectivity. The performance gap between techniques with and without RangeTrim generally decreases with increasing selectivity — perhaps because filters with higher selectivity tend to have range bounds that are not as conservative when compared with the a priori range bounds known to hold for the entire column.

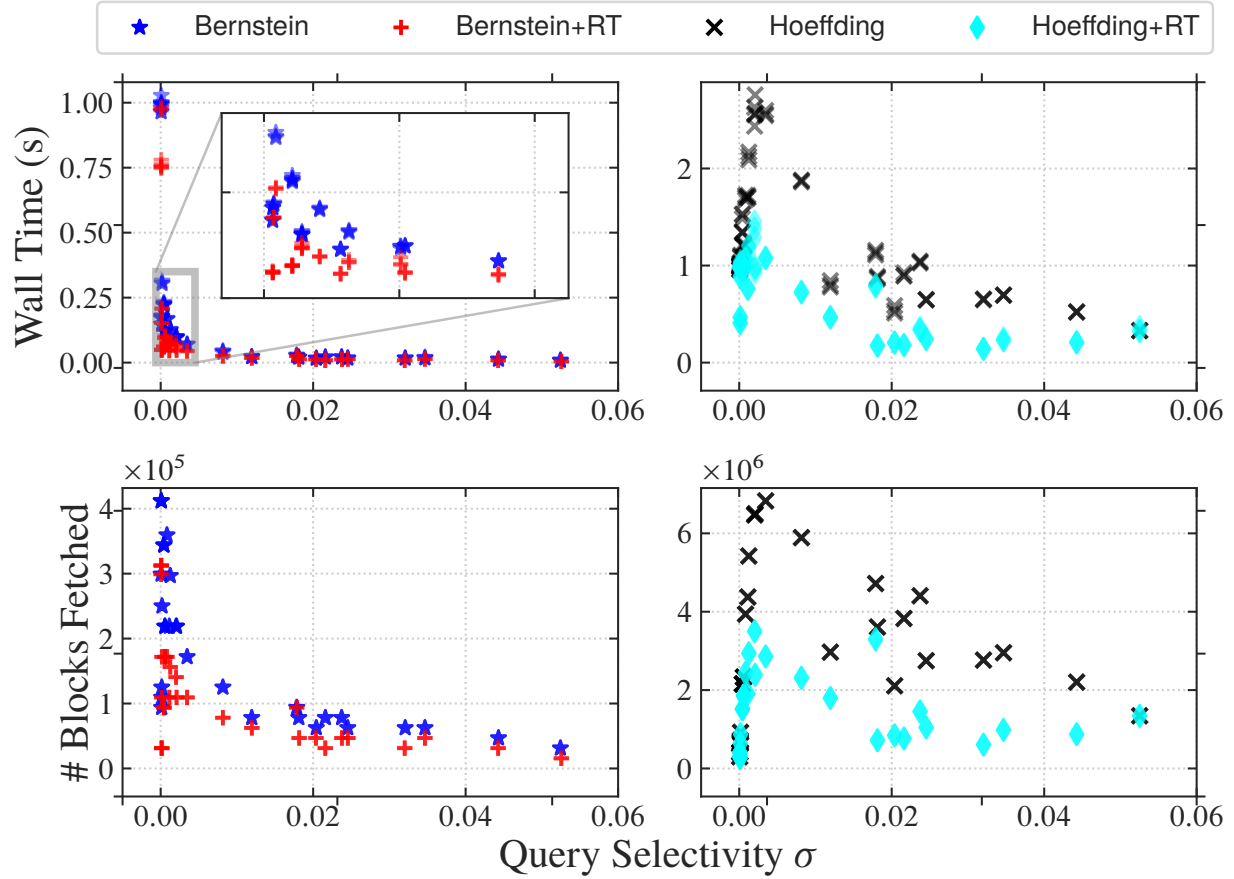


Figure 6.7: Effect of query selectivity on wall time and blocks fetched, for selectivity determined by varying the origin airport used to filter $F\text{-}q1[\varepsilon = .5]$.

ε for stopping condition ③.

Summary. For different upper bounds on relative error, the actual relative error in the query result is always within the requested error, for all error bounders applied to $F\text{-}q1$. The achieved relative error drops to 0 more quickly for the more conservative bounders Hoeffding and Hoeffding+RT as the requested error is decreased.

By varying the requested maximum relative error ε for query $F\text{-}q1$, we reveal its impact on the relative error achieved for various error bounders, shown in Figure 6.8(a). The main takeaways are that, for all error bounders, the achieved relative error generally decreases as the requested error bound decreases, with Hoeffding-based bounders dropping more quickly, as they are more conservative due to PMA.

HAVING threshold for stopping condition ④.

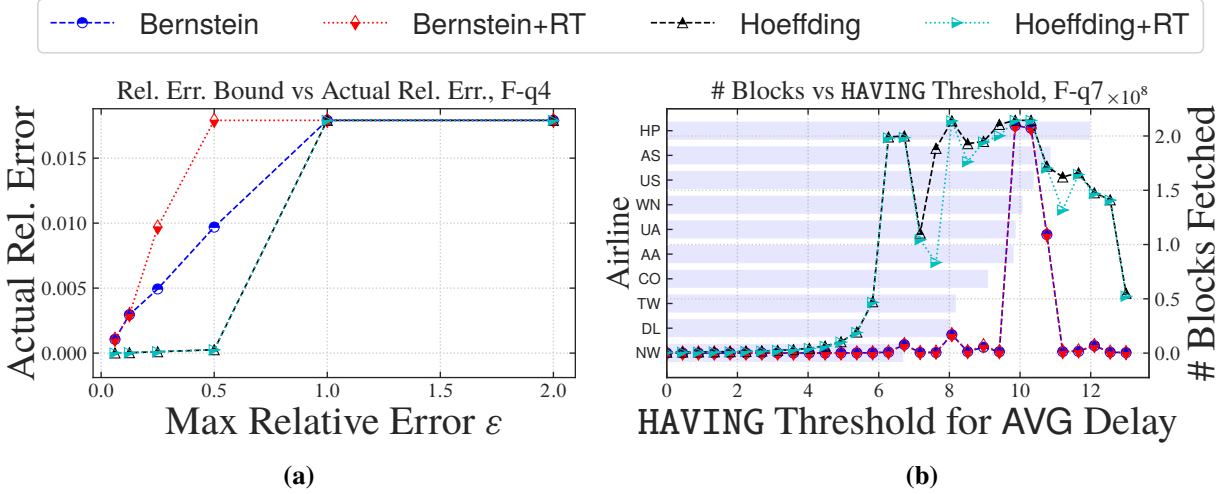


Figure 6.8: (a): Effect of requested maximum relative error ϵ on actual relative error achieved for F-q1. (b): Data required for different HAVING thresholds used in F-q2. The group aggregates for also displayed for comparison.

Summary. HAVING thresholds that are closer to group aggregates require more samples in order to achieve stopping condition ④, and Hoeffding-based error bounders in particular are more sensitive than Bernstein-based error bounders for the same threshold.

By varying the HAVING threshold used to filter groups / airlines post-aggregate in F-q2 and measuring its effect on the number of blocks fetched for a particular query, we reveal interesting data-dependent behavior impacted by the true aggregates for each airline, depicted in Figure 6.8(b). This figure also plots the group aggregates using a horizontal bar chart sharing the same x-axis as the HAVING threshold, revealing that it is “harder” to determine which side of the HAVING threshold a given group is if its aggregate is close to the threshold. Indeed, from Figure 6.8(b), we see that the initial thresholds near 0 are very easy for all groups, allowing for very fast termination. The first spike in number of blocks fetched occurs between 6 and 7, corresponding to the aggregate for airline NW. At and after this point, we see spikes in blocks fetched for both Hoeffding-based and Bernstein-based error bounders whenever the threshold approaches one or more airline aggregate values, although we note that Bernstein-based error bounders appear to be more robust, requiring the threshold to be much closer before they are adversely affected as compared to Hoeffding-based bounders.

Minimum departure time for F-q3.

Summary. As the minimum departure time is increased, the spread of average delay between airlines increases, making it easier to separate the two airlines with the minimum average delays

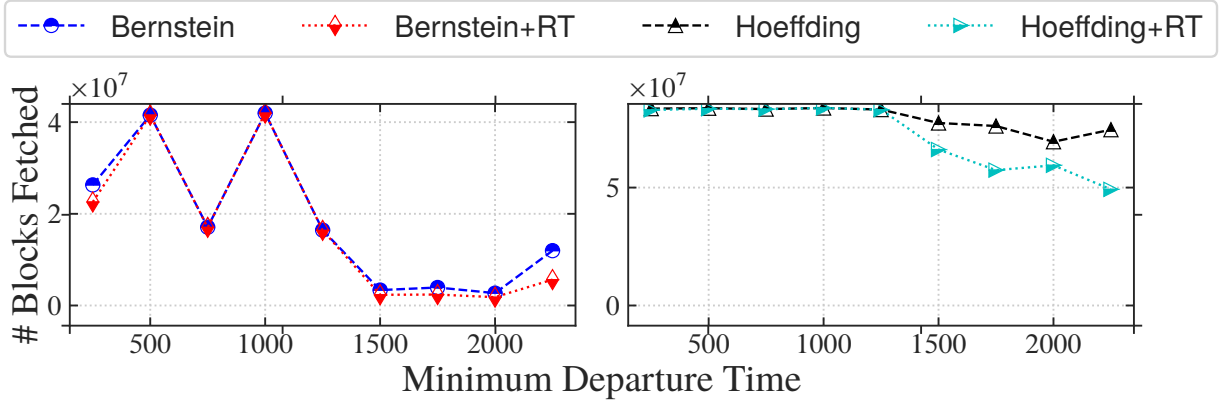


Figure 6.9: Effect of minimum departure time on blocks fetched for F-q3.

and achieve stopping condition ⑤ earlier. At the same time, termination becomes bottlenecked on sparse airlines, increasing the gap between similar bounders with and without RangeTrim.

By varying the minimum departure time `$min_dep_time` in F-q3, we reveal its impact on the number of blocks fetched for various error bounders, shown in Figure 6.9. This plot exhibits two interesting data-dependent behaviors worth unpacking. First, as the `$min_dep_time` increases, the variance in average delay between different airlines increases, perhaps because some airlines tend to have flights that are delayed more for later flights as compared with other airlines. This makes it easier to achieve stopping condition ⑤, since the average delays become more spread out with increasing minimum departure time, so we observe a decreasing trend in the number of blocks fetched. At the same time, as `$min_dep_time` increases, the selectivity of the various groups decreases. Since all the groups are sparser, the groups for which stopping condition ⑤ is bottlenecked are also sparser. Since we have an “easy” query (due to the higher variance between groups) for which sparse groups are bottlenecking termination, we tend to see a bigger performance gap between bounders with and without our RangeTrim technique.

Impact of ε on latency.

Summary. Figure 6.10 demonstrates that latency increases super-exponentially as ε decreases to 0, across all error bounders for F-q1[`$origin=ORD`].

By varying the maximum relative error bound ε , we reveal its impact on latency for various bounders in F-q1[`$origin=ORD`]. As depicted in Figure 6.10, latency increases super-exponentially with decreasing relative error threshold ε (note the log scale on the x-axis). From inspection of formulas for Hoeffding and Bernstein-style bounds, we would expect exponential behavior; the super-exponential behavior can be attributed to the conservative optional stopping procedure, which sacrifices some statistical efficiency. This figure illustrates the importance of being able to leverage

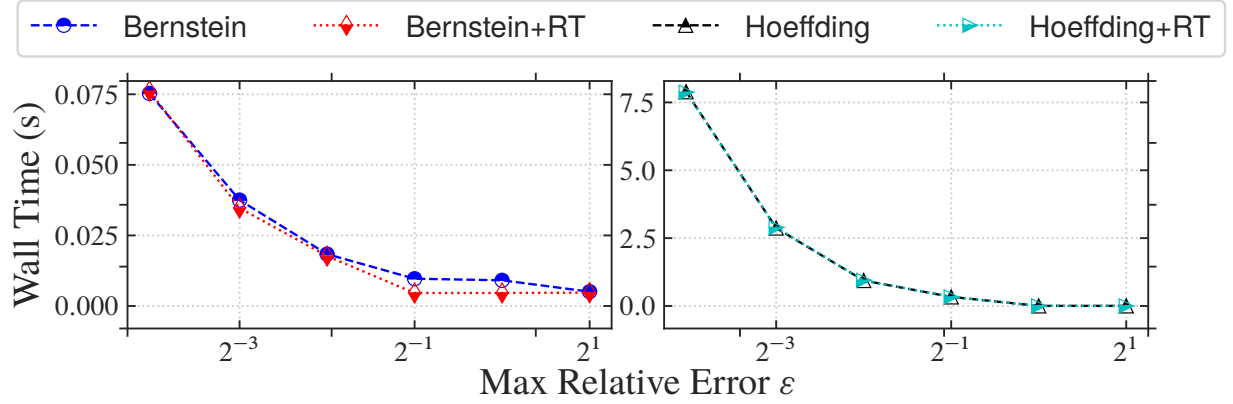


Figure 6.10: Effect of ε on wall time for $\text{F-q1}[\text{origin=ORD}]$

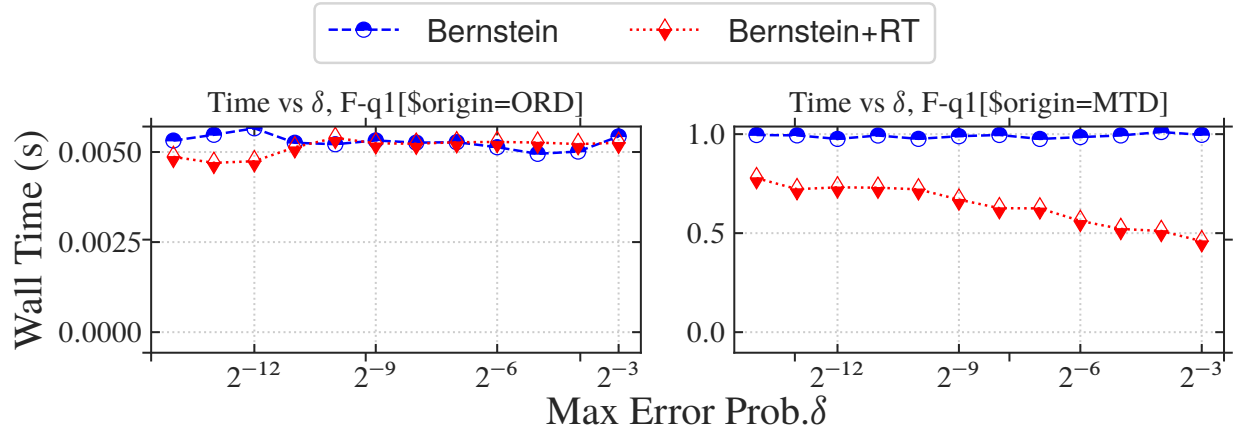


Figure 6.11: Effect of δ on wall time for $\text{F-q1}[\text{origin=ORD}]$ and $\text{F-q1}[\text{origin=MTJ}]$, respectively.

other stopping conditions besides that of condition ③ in the case of queries that do not actually need to make use of the aggregate but merely use it for downstream decision making, since such queries can sometimes tolerate very large relative errors.

Impact of δ on latency.

Summary. Figure 6.11 illustrates that latency is robust to small δ 's, with sublinear slowdowns for exponentially decreasing δ .

By varying δ , we reveal its impact on latency for bounders Bernstein and Bernstein+RT on $\text{F-q1}[\text{origin=ORD}]$ and $\text{F-q1}[\text{origin=MTJ}]$. In the case of a high-selectivity predicate (i.e., that for $\text{F-q1}[\text{origin=ORD}]$), we see virtually no impact of δ on latency; for this high-selectivity predicate, the query is able to terminate after the first bounds computation for both Bernstein and Bernstein+RT. For the sparser predicate in $\text{F-q1}[\text{origin=MTJ}]$, δ does not impact the number of

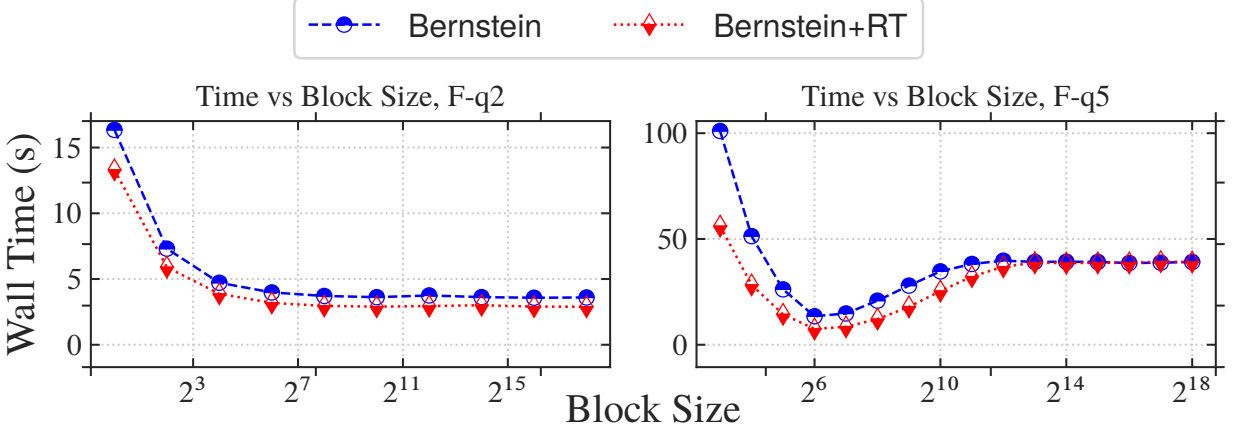


Figure 6.12: Effect of block size on wall time for F-q2 and F-q5, respectively.

rounds of bounds computation needed for Bernstein, although a larger δ does gradually decrease this quantity for Bernstein+RT, albeit insignificantly considering the exponential increase in δ . Overall, this illustrates that the dependence of each error bound on $\sqrt{\log(1/\delta)}$ translates to query latency that is highly robust in δ , motivating our choice of 10^{-15} as the default δ .

Impact of block size on latency.

Summary. Figure 6.12 suggests a quasiconvex relationship between latency and block size.

By varying the block size, we reveal its impact on latency for F-q2 and F-q5. For both of these queries, smaller block sizes are associated with higher latency due to poorer cache locality and more frequent bounds recomputation. Higher block sizes, however, lose out on benefits from block skipping. This is not an issue for F-q2, which has relatively few groups (and none of which are sparse); for F-q5, however, it impacts latency adversely before leveling off. The level-off in both cases can be attributed to saturating the bitmap indexes, since larger block sizes will be more likely to contain tuples passing each group’s filter.

6.6 SUMMARY

We categorized existing conservative error bounders in terms of two pathologies, PMA and PHOS, and developed a technique, RangeTrim, for eliminating PHOS from any range-based error bounder. We showed the advantage of using the empirical Bernstein-Serfling bounder in the context of a real system we are developing, FASTFRAME, that accelerates approximate queries significantly over a Hoeffding-Serfling-based error bounder, which suffers from PMA. We furthermore showed that augmenting this error bounder with our RangeTrim technique leads to an additional 2× in the best

case, without ever hurting performance in the worst case. By implementing our distribution-aware techniques in the context of FASTFRAME, which is aware of practical considerations such as locality, optional stopping, and block skipping in order to prioritize groups that require more samples in order to facilitate early termination, we demonstrate significant speedups (on the order of $10\times$ over both exact processing and traditional techniques based on Hoeffding) without losing guarantees. This suggests a viable path toward practical with-guarantees data analytics.

With this chapter, we conclude our study of safety and interactivity of specific data science tasks; next, we consider techniques for improving safety of a common *data science environment*, the computational notebook, without compromising on interactivity.

CHAPTER 7: APPROXIMATE LINEAGE FOR SAFE NOTEBOOK INTERACTIONS

In this chapter, we describe `NBSAFETY`, a custom Jupyter kernel that uses builds approximate lineage of the state in a computational notebook session and uses this lineage to warn users of potential errors before they occur. After motivating the problem, we describe `NBSAFETY`’s design and how it uses trace-once semantics to keep the overhead of lineage maintenance low, while still retaining enough lineage to be useful on notebooks as they are used in practice. We then describe `NBSAFETY`’s program analysis component and how it is used to provide warnings, before conducting an empirical study on a dataset of real notebook session traces.

7.1 MOTIVATION

Computational notebooks such as Jupyter [25] provide a flexible medium for developers, scientists, and engineers to complete programming tasks interactively. Notebooks, like simpler predecessor read-eval-print-loops (REPLs), do not terminate after executing, but wait for the user to give additional instructions while keeping intermediate programming state in memory. Notebooks, however, are distinguished from REPLs by three key features:

1. The atomic unit of execution in notebooks is the *cell*, composed of a sequence of one or more programming statements, rather than a single programming statement;
2. Notebooks allow users to easily refer back to previous cells to make edits and potentially re-execute; and
3. Notebooks typically allow code and documentation to be interspersed, following the *literate programming* [187] paradigm.

As a result, the IPython Notebook project [188], and its successor, Project Jupyter [25], have both grown rapidly in popularity. These projects decouple the server-side *kernel* (responsible for running user code) from a browser-accessible client side (providing the user interface). Since Jupyter uses familiar web technologies to implement the layer that facilitates communication between the UI and the kernel, it crucially allows users to run notebooks on any platform. Furthermore, Jupyter’s decoupled architecture allows users to leverage powerful server computing resources from modest client-side hardware, particularly useful for coping with the ever-increasing scale of modern dataset sizes.

These key features, along with the natural ergonomics of interactive computing offered by the notebook interface, have led to an explosion in Jupyter’s usage. With more than 4.7 million notebooks on GitHub as of March 2019 [189] and with hosted solutions offered by a plethora of data analytics companies, Jupyter has been called “data scientists’ computational notebook of

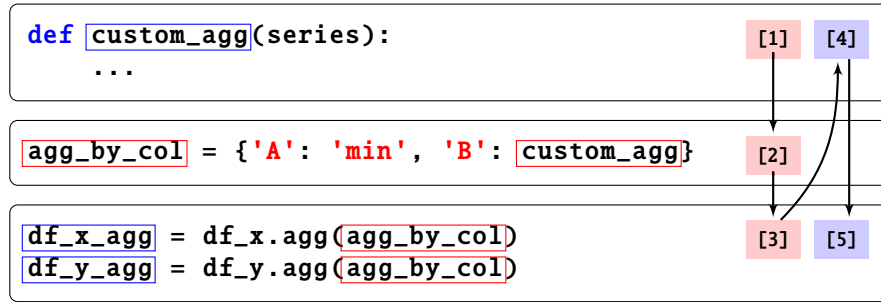


Figure 7.1: Example sequence of notebook interactions leading to a stale symbol usage. Symbols with timestamps ≤ 3 (resp. > 3) are shown with blue (resp. red) borders.

choice” [87], and was recognized by the ACM Software System award in 2018 [190]. We focus on Jupyter here due to its popularity, but our ideas are applicable to computational notebooks in general.

Despite the tighter feedback enjoyed by users of computational notebooks, and, in particular, by users of Jupyter, notebooks have a number of drawbacks when used for more interactive and exploratory analysis. Compared to conventional programming environments, interactions such as *out-of-order cell execution*, *cell deletion*, and *cell editing and re-execution* can all complicate the relationship between the code visible on screen and the resident notebook state. Managing interactions with this hidden notebook state is thus a burden shouldered by users, who must remember what they have done in the past, since these past interactions cannot in general be reconstructed from what is on the screen.

Illustration. Consider the sequence of notebook interactions depicted in Figure 7.1. Each rectangular box indicates a cell, the notebook’s unit of execution. The user first defines a custom aggregation function that, along with `min`, will be applied to two dataframes, `df_x` and `df_y`, and executes it as cell [1]. Since both aggregations will be applied to both dataframes, the user next gathers them into a function dictionary in the second cell (executed as cell [2]). After running the third cell, which corresponds to applying the aggregates to `df_x` and `df_y`, the user realizes an error in the logic of `custom_agg` and goes back to the first cell to fix the bug. They re-execute this cell after making their update, indicated as [4]. However, they forget that the old version of `custom_agg` still lingers in the `agg_by_col` dictionary and rerun the third cell (indicated as [5]) without rerunning the second cell. We deem this an *unsafe* execution, because the user intended for the change to `agg_by_col` to be reflected in `df_agg_x` and `df_agg_y`, but it was not. Upon inspecting the resulting dataframes `df_x_agg` and `df_y_agg`, the user may or may not realize the error. In the best case, user may identify the error and rerun the second cell. In the worst case,

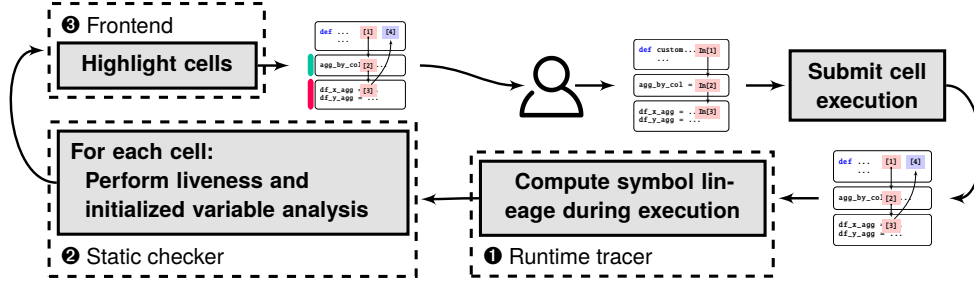


Figure 7.2: NBSAFETY workflow with architectural components.

users may be deceived into thinking that their change had no effect, with the original error then propagating throughout the notebook.

This example underscores the inherent difficulty in manually managing notebook state, inspiring colorful criticisms such as a talk titled “I Don’t Like Notebooks” presented at JupyterCon 2018 [26]. In addition to the frustration that users experience when spending valuable time debugging state-related errors, such bugs can lead to invalid research results and hinder reproducibility, inspiring the claim that one must “restart and run all or it didn’t happen” [87] if presenting results via a notebook medium. Thus, in this chapter we develop techniques to automatically identify and prevent potentially unsafe cell executions, without sacrificing existing familiar notebook semantics.

7.2 ARCHITECTURE OVERVIEW

In this section, we give an overview of NBSAFETY’s components and how they integrate into the notebook workflow.

Overview. NBSAFETY integrates into a notebook workflow according to Figure 7.2. As depicted, all components of NBSAFETY are invoked upon each and every cell execution. When the user submits a request to run a cell, the tracer (1, Section 7.3) instruments the executed cell, updating lineage metadata associated with each variable as each line executes. Once the cell finishes execution, the checker (2, Section 7.4) performs *liveness analysis* [191] (a standard technique for finding non-redefined symbols in a program) and *initialized variable analysis* [192] (discussed in Section 7.4.3) for every cell in the notebook. By combining the results of these analyses with the lineage metadata computed by the tracer, the frontend (3, Section 7.5) is able to highlight cells that are unsafe due to staleness issues of the form seen in Figure 7.1, as well as cells that resolve such staleness issues.

1 Tracer. The NBSAFETY tracer maintains dataflow dependencies for each symbol that appears in the notebook in the form of lineage metadata. It leverages Python’s built-in tracing capabilities [193], which allows it to run custom code upon four different kinds of events: (i) line events,

when a line starts to execute; (ii) `call` events, when a function is called, (iii) `return` events, when a function returns, and (iv) `exception` events, when an exception occurs.

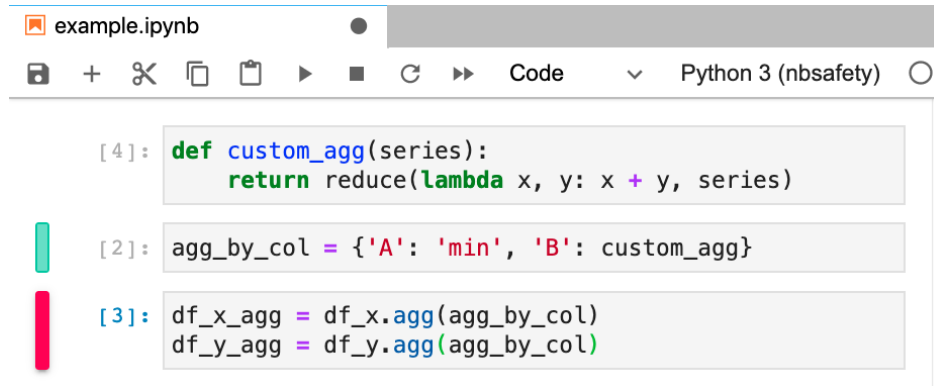
To illustrate its operation, consider that, the first time c_3 in Figure 7.1 is executed, symbols `df_agg_x` and `df_agg_y` are undefined. Before the first line runs, a `line` event occurs, thereby trapping into the tracer. The tracer has access to the line of code that triggered the `line` event and parses it as an `Assign` statement in Python’s grammar, followed by a quick static analysis to determine that the symbols `df_x` and `agg_by_col` appear on the right hand side of the assignment (i.e., these symbols appear in `USE[R.H.S. of the Assign]`). Thus, these two will be the dependencies for symbol `df_agg_x`. Since c_3 is the third cell executed, the tracer furthermore gives `df_agg_x` a timestamp of 3. Similar statements hold for `df_agg_y` once the second line executes.

❷ **Checker.** The `NBSAFETY` static checker performs two kinds of program analysis: (i) *liveness analysis*, and (ii) *initialized variable analysis*. The `NBSAFETY` liveness checker helps to detect safety issues by determining which cells have live references to stale symbols. For example, in Figure 7.1, `agg_by_col`, which is stale, is live in c_3 —this information can be used to warn the user before they execute c_3 . Furthermore, the initialized checker serves as a key component for efficiently computing resolutions to staleness issues, as we later show in Sections 7.4 and 7.5.

❸ **Frontend.** The `NBSAFETY` frontend uses the results of the static checker to highlight cells of interest. For example, in Figure 7.3, which depicts the original example from Figure 7.1 (but before the user submits c_3 for re-execution), c_3 is given a **staleness warning** highlight to warn the user that re-execution could have incorrect behavior due to staleness issues. At the same time, c_2 is given a **cleanup suggestion** highlight, because rerunning it would resolve the staleness in c_3 . The user can then leverage the extra visual cues to make a more informed decision about which cell to next execute, possibly preferring to execute cells that resolve staleness issues before cells with staleness issues.

Design Philosophy. In contrast with systems that automatically resolve staleness, such as Datalore [14] or Nodebook [13], `NBSAFETY` is designed to be as non-intrusive as possible, while providing useful information. `NBSAFETY` thus only attempts to make a “passive observer” guarantee to ensure that existing notebook semantics are preserved, via tracing and static analyses that monitor, and do not alter, notebook behavior. We make this decision partially because program analysis in a dynamic language like Python is notoriously difficult; as such, `NBSAFETY`’s analysis components are unable to make any formal guarantees regarding soundness or completeness, instead opting for a “best-effort” attempt to provide the user with useful information.

Although the guarantee `NBSAFETY` offers regarding preservation of semantics may seem weak at first glance, we show later (§7.6.5) that both Datalore and Nodebook crash on reasonable programs due to their intrusiveness. Furthermore, we also show how existing semantics lend themselves to



```
[4]: def custom_agg(series):  
      return reduce(lambda x, y: x + y, series)  
  
[2]: agg_by_col = {'A': 'min', 'B': custom_agg}  
  
[3]: df_x_agg = df_x.agg(agg_by_col)  
      df_y_agg = df_y.agg(agg_by_col)
```

Figure 7.3: NBSAFETY highlights unsafe cells with staleness warnings and cells that resolve staleness issues with cleanup suggestions.

multiverse analyses [194], and point to specific examples of such analyses in our experiments. Finally, NBSAFETY is not necessarily intended to be a substitute for systems like Dataflow notebooks [12] that allow users to give names to cell outputs and explicitly reference them; the two techniques can complement each other, as the issues faced by notebook users do not automatically disappear by using Dataflow notebooks.

Overall, each of NBSAFETY’s three key components play crucial roles in helping users avoid and resolve unsafe interactions due to staleness issues without compromising existing notebook program semantics. We describe each component in the following sections.

7.3 LINEAGE TRACKING

In this section, we describe how NBSAFETY traces cell execution in order to maintain symbol lineage metadata, and how such metadata aids in the detection and resolution of staleness issues. We begin by introducing helpful terminology and formalizing our notion of staleness beyond the intuition we gave in Figure 7.1 of Section 7.1.

7.3.1 Preliminaries

We begin defining our use of the term *symbol*.

Definition 7.1 [Symbol]. A symbol is any piece of data in notebook scope that can be referenced by a (possibly qualified) name.

For example, if `lst` is a list with 10 entries, then `lst`, `lst[0]`, and `lst[8]` are all symbols. Similarly, if `df` is a dataframe with a column named “col”, then `df` and `df.col` are both symbols.

Symbols can be thought of as a generalized notion of variables that allow us treat different nameable objects in Python’s data model in a unified manner.

NBSAFETY augments each symbol with additional lineage metadata in the form of *timestamps* and *dependencies*.

Definition 7.2 [Timestamp]. *A symbol’s timestamp is the execution counter of the cell that most recently modified that symbol. Likewise, a cell’s timestamp is the execution counter corresponding to the most recent time that cell was executed.*

For a symbol s or a cell c , we denote its timestamp as $ts(s)$ or $ts(c)$, respectively. For example, letting c_1 , c_2 , and c_3 denote the three cells in [Figure 7.1](#), we have that $ts(custom_agg) = ts(c_1) = 4$, since `custom_agg` is last updated in c_1 , which was executed at time 4.

Definition 7.3 [Dependencies]. *The dependencies of symbol s are those symbols that contributed to s ’s computation via direct dataflow.*

In [Figure 7.1](#), `agg_by_col` depends on `custom_agg`, while `df_x_agg` depends on `df_x` and `custom_agg`. We denote the dependencies of s with $Par(s)$. Similarly, if $t \in Par(s)$, then $s \in Chd(t)$.

A major contribution of NBSAFETY is to highlight cells with unsafe usages of *stale symbols*, which we define recursively as follows:

Definition 7.4 [Stale symbols]. *A symbol s is called stale if there exists some $s' \in Par(s)$ such that $ts(s') > ts(s)$, or s' is itself stale; that is, s has a parent that is either itself stale or more up-to-date than s .*

In [Figure 7.1](#), symbol `agg_by_col` is stale, because $ts(agg_by_col) = 2$, but $ts(custom_agg) = 4$. Staleness gives us a rigorous conceptual framework upon which to study the intuitive notion that, because `custom_agg` was updated, we should also update its child `agg_by_col` to prevent counterintuitive behavior.

We now draw on these definitions as we describe how NBSAFETY maintains lineage metadata while tracing cell execution.

7.3.2 Lineage Update Rules

NBSAFETY attempts to be non-intrusive when maintaining lineage with respect to the Python objects that comprise the notebook’s state. To do so, we avoid modifying the Python objects created by the user, instead creating “shadow” references to each symbol. NBSAFETY then takes a hybrid dynamic / static approach to updating each symbol’s lineage. After each statement has finished executing, the tracer inspects the AST node for the executed statement and performs a lineage update according to the rules shown in [Table 7.1](#).

AST Node	Example	Rule
Assign (target in RHS)	$a = e$ $a = a + e$	$\text{Par}(a) = \text{USE}[e]$ $\text{Par}(a) = \text{Par}(a) \cup \text{USE}[e]$
Assign with Call	$a = f(e)$	$\text{Par}(a) = \text{USE}[e] \cup \text{RET}[f]$
AugAssign	$a += e$	$\text{Par}(a) = \text{Par}(a) \cup \text{USE}[e]$
For	for a in e :	$\text{Par}(a) = \text{USE}[e]$
FunctionDef	def $f(a=e)$:	$\text{Par}(f) = \text{USE}[e]$
ClassDef	class $c(e)$:	$\text{Par}(c) = \text{USE}[e]$

Table 7.1: Subset of lineage rules used by the NBSAFETY tracer

Example. Suppose the statement

$$\text{gen} = \text{map}(\text{lambda } x: f(x), \text{foo} + [\text{bar}]) \quad (7.1)$$

has just finished executing. Using rule 1 of [Table 7.1](#), the tracer will then statically analyze the right hand side in order to determine

$$\text{USE}[\text{map}(\text{lambda } x: f(x), \text{foo} + [\text{bar}])] \quad (7.2)$$

which is the set of used symbols that appear in the RHS. In this case, the aforementioned set is $\{f, \text{foo}, \text{bar}\}$ — everything else is either a Python built-in (`map`, `lambda`), or an unbound symbol (i.e. in the case of the lambda argument x). The tracer will thus set

$$\text{Par}(\text{gen}) = \{f, \text{foo}, \text{bar}\} \quad (7.3)$$

and will also set $\text{ts}(\text{gen})$ to the current cell’s execution counter.

Handling Function Calls and Returns. Recall that, in Python, a function may “capture” symbols defined in external scope by referencing them. In this case, the lineage update rule for a function call needs to be aware of the symbols referenced by the function’s return statement. As such, the NBSAFETY tracer saves these symbols when it encounters a return statement, and loads them upon encountering a `return` event, so that they are available for use with lineage update rules, e.g., the third entry of [Table 7.1](#).

Rationale for Tracing. Given that the tracer is already performing some static analysis as each Python statement executes, a natural question is: why should we trace cell execution at all, instead of performing static analysis on an entire cell in order to make lineage updates? The answer is that, when a cell executes, a relatively small number of control flow paths may be taken, and a purely static approach must consider them all in order to be conservative. This may cause each $\text{Par}(s)$ to

be much larger than necessary, or to be unnecessarily overwritten if, e.g., s is assigned in some unexecuted control flow path. This can happen due to, e.g., untaken branches, or if an exception is thrown mid-cell. Indeed, we attempted to infer lineage updates statically in an earlier version of `NBSAFETY`, but found this to be too coarse-grained in order to derive real benefits.

Staleness Propagation. We already saw that the tracer annotates each symbol’s shadow reference with timestamp and lineage metadata. Additionally, it tracks whether each symbol is stale, as this cannot be inferred solely from timestamp and lineage metadata. To see why, recall the definition of staleness: a symbol s is stale if it has a more up-to-date parent (i.e., an $s' \in \text{Par}(s)$ with $\text{ts}(s') > \text{ts}(s)$), *or if it has a stale parent*, precluding the ability to determine staleness locally. Thus, when s is updated, we perform a depth first search starting from each child $c \in \text{Chd}(s)$ in order to propagate the “staleness” to all descendants.

Fine-Grained Lineage for Attributes and Subscripts. Recall that the `NBSAFETY` tracer attempts to infer parent symbols statically when making lineage updates. While this is possible in many cases, there are limits to a purely static approach. For example, the statement “ $s = a.b().c$ ” is valid Python code, but, in general, it is impossible to statically determine what $a.b()$ returns. As such, we must again rely on tracing to do so. Unfortunately, Python’s built-in tracing abilities operate at the granularity of code lines, so that a `return` event does not tell us the point within a line to which control returns. Thus, the tracer rewrites all `Attribute` nodes in the statement’s AST, so that the earlier example will actually run as follows:

$$s = \text{trace}(\text{trace}(a, 'b').b(), 'c').c \quad (7.4)$$

The `trace` function will first determine that the symbol name b is referenced within symbol a ’s namespace, setting the current RHS parent symbol to $a.b$. However, after the call event, the tracer will update this to the c symbol in the namespace of the return value of $a.b()$, so that the true parent symbol is pinpointed.

Subscripts are handled analogously to attributes.

Handling External Libraries. `NBSAFETY` assumes that imported libraries do not have access to notebook state. Thus, when the tracer observes a call into library code, it simply halts tracing, resuming once control returns to the notebook.

Bounding Lineage Overhead. Consider the following cell:

```
x = 0
for i in random.sample(range(10**7), 10**5) + [42]:
    x += lst[i]
```

Figure 7.4: Example wherein exact lineage would not compress.

In order to maintain lineage metadata for symbol `x` to 100% correctness, we would need to somehow indicate that `Par(x)` contains `lst[i]` for all 10^5 random indices `i` (as well as for `lst[42]`). It is impossible to maintain acceptable performance in general under these circumstances. Potential workarounds include *conservative* approximations, as well as *lossy* approximations. For example, as a conservative approximation, we could instead specify that `x` depends on `lst`, with the implication that it also depends on everything in `lst`’s namespace. However, this will cause `x` to be incorrectly classified as stale whenever `lst` is mutated, e.g., if a new entry is appended.

Thus, `NBSAFETY` makes a compromise and sacrifices some correctness by *only instrumenting each AST statement the first time it executes*. In this case, after the cell executes, `Par(x)` will have a single entry: `lst[0]`. These *trace-once semantics* help to ensure that lineage metadata only grows in proportion to the amount of text in the user’s notebook.

Note that this approach can lead to some false negatives when determining which symbols are stale. In the example above, if the user makes a point update to, e.g., `lst[42]`, `x` should technically be considered stale, but will fail to be updated as so since `lst[42] ∉ Par(x)`. We consider this tradeoff worthwhile, as we found that it covered most common notebook usage patterns in our experiments, and furthermore found that performance suffered greatly without it.

Garbage Collection. Each symbol’s shadow metadata maintains a weak reference to the symbol. When a Python object is deleted, e.g., because the user executed a `del` statement, or because it was garbage collected by Python’s built-in garbage collector, a callback associated with the weak reference executes. This callback tells the metadata to delete itself (including from any `Par(·)` or `Chd(·)` sets), thereby ensuring that lineage metadata does not accumulate without bound over long notebook sessions.

Handling Aliases. Since multiple symbols can reference the same object in Python, we need a mechanism to propagate mutations to some object to all symbols that reference it. For example, if both `x` and `y` refer to the same list, and then this list is appended to, both `x` and `y` should have their timestamps bumped. `NBSAFETY` accomplishes this by creating a registry that maps objects to all symbols referencing them.

7.4 LIVENESS AND INVERSE LIVENESS

Algorithm 7.1: Liveness checker

Input: Cell c , CFG G that stores successors $\text{succ}[s]$ for each stmt $s \in c$
Output: $\text{LIVE}(c)$

```

1  foreach stmt  $s \in c$  do
2       $\text{LIVE}_{out}[s] \leftarrow \emptyset$ ;
3       $\text{LIVE}_{in}[s] \leftarrow \emptyset$ ;
4       $\text{USE}[s] \leftarrow \{\text{referenced symbols in } s\}$ ;
5       $\text{DEF}[s] \leftarrow \{\text{assigned symbols or defined functions in } s\}$ ;
6  end
7   $\text{LIVE}_{in}[G_{exit}] \leftarrow \emptyset$ ;
8  repeat
9      foreach stmt  $s \in c$  do
10          $\text{LIVE}'_{out}[s] \leftarrow \text{LIVE}_{out}[s]$ ;
11          $\text{LIVE}'_{in}[s] \leftarrow \text{LIVE}_{in}[s]$ ;
12          $\text{LIVE}_{out}[s] \leftarrow \bigcup_{s' \in \text{succ}[s]} \text{LIVE}_{in}[s']$ ;
13          $\text{LIVE}_{in}[s] \leftarrow \text{USE}[s] \cup (\text{LIVE}_{out}[s] - \text{DEF}[s])$ ;
14     end
15 until  $\text{LIVE}_*[s] = \text{LIVE}'_*[s]$  for both  $* \in \{in, out\}$ ,  $\forall s \in c$ ;
16 return  $\text{LIVE}_{in}[G_{entry}]$ ;
```

Algorithm 7.2: Initialized variable checker

Input: Cell c , CFG G that stores predecessors $\text{pred}[s]$ for each stmt $s \in c$
Output: $\text{DEAD}(c)$

```

1  foreach stmt  $s \in c$  do
2       $\text{DEAD}_{in}[s] \leftarrow \emptyset$ ;
3       $\text{DEAD}_{out}[s] \leftarrow \mathcal{U}$ ;
4       $\text{USE}[s] \leftarrow \{\text{referenced symbols in } s\}$ ;
5       $\text{DEF}[s] \leftarrow \{\text{assigned symbols or defined functions in } s\}$ ;
6  end
7   $\text{DEAD}_{out}[G_{entry}] \leftarrow \emptyset$ ;
8  repeat
9      foreach stmt  $s \in c$  do
10          $\text{DEAD}'_{in}[s] \leftarrow \text{DEAD}_{in}[s]$ ;
11          $\text{DEAD}'_{out}[s] \leftarrow \text{DEAD}_{out}[s]$ ;
12          $\text{DEAD}_{in}[s] \leftarrow \bigcap_{s' \in \text{pred}[s]} \text{DEAD}_{out}[s']$ ;
13          $\text{DEAD}_{out}[s] \leftarrow (\text{DEF}[s] - \text{USE}[s]) \cup \text{DEAD}_{in}[s]$ ;
14     end
15 until  $\text{DEAD}_*[s] = \text{DEAD}'_*[s]$  for both  $* \in \{in, out\}$ ,  $\forall s \in c$ ;
16 return  $\text{DEAD}_{out}[G_{exit}]$ ;
```

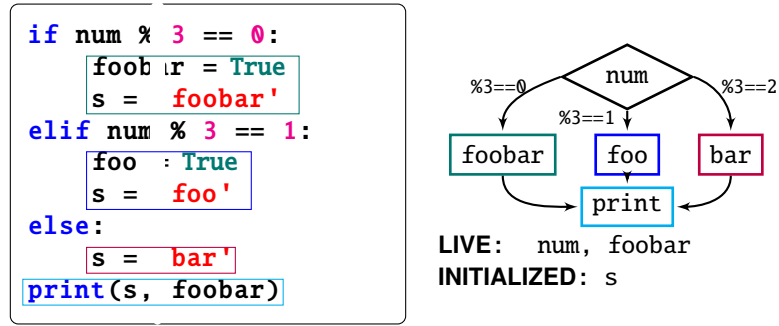


Figure 7.5: Example liveness and initialized variable analysis.

In this section, we describe the program analysis component of NBSAFETY’s backend. The *checker* performs liveness analysis [191], and a lesser-known program analysis technique called initialized variable analysis, or definite assignment analysis [192]. These techniques are crucial for efficiently identifying which cells are unsafe to execute due to stale references, as well as which cells help resolve staleness issues. We begin with background before discussing the connection between these techniques and staleness detection and resolution.

7.4.1 Background

Liveness analysis [191] is a program analysis technique for determining whether the value of a variable at some point is used later in the program. Although traditionally used by compilers to, for example, determine how many registers need to be allocated at some point during program execution, we use it to determine whether a cell has references to stale symbols. We also show (§7.5) how initialized variable analysis [192], a technique traditionally used by IDEs and linters to detect potentially uninitialized variables, can be used to efficiently determine which cells to resolve staleness.

Example. In Figure 7.5, symbols `num` and `foobar` are live at the top of the cell, since the value for each at the top of the cell can be used in some path of the control flow graph (CFG). In the case of `num`, the (unmodified) value is used in the conditional. In the case of `foobar`, while one path of the CFG modifies it, the other two paths leave it unchanged by the time it is used in the `print` statement; hence, it is also live at the top of the cell. The symbol that is not live at cell start is `foo`, since it is only ever assigned and never used, and `s`, since every path in the CFG assigns to `s`. We call symbols such as `s` that are assigned in every path of the CFG *dead* once they reach the end of the cell.

7.4.2 Cell Oriented Analysis

We now describe how we relate liveness, which is traditionally applied in the context of a single program, to a notebook environment. In brief, we treat each cell as if it is an individual program when performing various program analyses. We formalize these notions below.

Definition 7.5 [Live symbols]. *Given a cell c and some symbol s , we say that s is live in c if there exists some execution path in c in which the value of s at the start of c 's execution is used later in c .*

In other words, s is live in c if, treating c as a standalone program, s is live in the traditional sense at the start of c . We already saw in [Figure 7.5](#) that the live symbols in the example cell are `num`, `fiz`, and `buz`. For a given cell c , we use $\text{LIVE}(c)$ to denote the live symbols in c .

We are also interested in *dead symbols* that are (re)defined in every branch by the time execution reaches the end of a given cell c .

Definition 7.6 [Dead symbols]. *Given a cell c and some symbol s , we say that s is dead in c if, by the time control reaches the end of c , every possible path of execution in c overwrites s in a manner independent of the current value of s .*

Denoting such symbols as $\text{DEAD}(c)$, we will see in [Section 7.5](#) the role they play in assisting in the resolution of staleness issues.

Staleness and Freshness of Live Symbols in Cells. Recall that symbols are augmented with additional lineage and timestamp metadata computed by the tracer ([§7.3](#)). We can thus additionally refer to the set $\text{STALE}(c) \subseteq \text{LIVE}(c)$, the set of stale symbols that are live in c . When this set is nonempty, we say that cell c itself is stale:

Definition 7.7 [Stale cells]. *A cell c is called stale if there exists some $s \in \text{LIVE}(c)$ such that s is stale; i.e., c has a live reference to some stale symbol.*

A major contribution of `NBSAFETY` is to identify cells that are stale and preemptively warn the user about them.

Note that a symbol can be stale regardless of whether it is live in some cell. Given a particular cell c , we can also categorize symbols according to their lineage and timestamp metadata as they relate to c . For example, when a non-stale symbol s that is live in c is more “up-to-date” than c , then we say that it is *fresh* with respect to c :

Definition 7.8 [Fresh symbols]. *Given a cell c and some symbol s , we say that s is fresh w.r.t. c if (i) s is not stale, and (ii) $\text{ts}(s) > \text{ts}(c)$.*

We can extend the notion of fresh symbols to cells just as we did for stale symbols and stale cells:

Definition 7.9 [Fresh cells]. *A cell c is called fresh if it (i) it is not stale, and (ii) it contains a live reference to one or more fresh symbols; that is, $\exists s \in \text{LIVE}(c)$ such that s is fresh with respect to c .*

Example. Consider a notebook with three cells run in sequence, with code $a=4$, $b=a$, and $c=a+b$, respectively, and suppose the first cell is updated to be $a=5$ and rerun. The third cell contains references to a and b , and although a is fresh, b is stale, so the third cell is not fresh, but stale. On the other hand, the second cell contains a live reference to a but no live references to b , and is thus fresh.

As we see in our experiments (§7.6), fresh cells are oftentimes cells that users wish to re-execute; another major contribution of NBSAFETY is therefore to automatically identify such cells. In fact, in the above example, rerunning the second cell resolves the staleness issue present in the first cell. That said, running any other cell that assigns to b would also resolve the staleness issue, so staleness-resolving cells need not necessarily be fresh. Instead, fresh cells can be thought of as resolving staleness in cell output, as opposed to resolving staleness in some symbol. We study such staleness-resolving cells next.

Cells that Resolve Staleness. We have already seen how liveness checking can help users to identify stale cells. Ideally, we should also identify cells whose execution would “freshen” the stale variables that are live in some cell c , thereby allowing c to be executed without potential errors due to staleness. We thus define *refresher cells* as follows:

Definition 7.10 [Refresher cells]. *A non-stale cell c_r is called refresher if there exists some other stale cell c_s such that*

$$\text{STALE}(c_s) - \text{STALE}(c_r \oplus c_s) \neq \emptyset \quad (7.5)$$

where $c_r \oplus c_s$ denotes the concatenation of cells c_r and c_s . That is, the result of merging c_r and c_s together has fewer live stale symbol references than does c_s alone.

Intuitively, if we were to submit a refresher cell for execution, we would reduce the number of stale symbols live in some other cell (possibly to 0). Note that a refresher cell may or may not be fresh.

In addition to identifying stale and fresh cells, a final major contribution of NBSAFETY is the *efficient* identification of refresher cells. We will see in Section 7.5 that scalable computation of such cells requires initialized analysis to compute dead symbols, which we describe in detail next.

7.4.3 Initialized Variable Analysis

Recall that we use liveness analysis to find “live” symbols whose values at the start of each cell contribute to the computation performed in the cell, and we use inverse liveness to find “dead” symbols whose values at the end of the each will have definitely been overwritten by the time control reaches the end of the cell. A working knowledge of traditional liveness analysis is a prerequisite for understanding our inverse liveness technique; we refer the reader to, e.g., Aho et al. [191] for any review necessary.

Dataflow Equations. Inverse liveness is a fixed-point method for solving the following set of dataflow equations:

$$DEAD_{out}[s] = (DEF[s] - USE[s]) \cup DEAD_{in}[s] \quad (7.6)$$

$$DEAD_{in}[s] = \bigcap_{s' \in \text{predecessors of } s} DEAD_{out}[s'] \quad (7.7)$$

That is, a symbol is dead in statement s (i) if it is defined as a function, (ii) if it appears on the left hand side of an assignment (but not the right hand side), or (iii) if it is dead in all predecessor statements in the control flow graph.

Intuition. Traditional liveness analysis initializes each statement at the minimum point of a lattice (i.e., the empty set). Each statement's set of used symbols ($USE[s]$) are then iteratively propagated in the reverse direction of control until a fixed point is reached, thereby solving the following set of dataflow equations:

$$LIVE_{in}[s] = USE[s] \cup (LIVE_{out}[s] - DEF[s]) \quad (7.8)$$

$$LIVE_{out}[s] = \bigcup_{s' \in \text{successors of } s} LIVE_{in}[s'] \quad (7.9)$$

If the live variables propagated during liveness checking can be thought of as electrons, then the dead variables propagated during inverse liveness can be thought of as “holes”, using a metaphor from electronics. Nearly every decision made over the course of inverse liveness analysis is the “inverse” of decisions made during liveness analysis. For example, in inverse liveness, each statement's set of dead symbols is initialized to *everything* (i.e., at lattice maximum), and inverse liveness uses set intersection (\cap) as the lattice meet operator instead of set union (\cup) when propagating dead variables between statements, since symbols can be dead at cell bottom only if they are overwritten in *every* branch of control.

Comparing Liveness and Inverse Liveness. A pseudocode description of our inverse liveness checker is given in [Algorithm 7.2](#). We also give a description of a textbook liveness checker to the left in [Algorithm 7.1](#) for contrast. In particular, we note that the two algorithms are nearly identical excepting a few key differences:

- On [line 3](#), $DEAD_{out}$ is initialized to *every* symbol, instead of \emptyset ;
- On [line 12](#) of [Algorithm 7.2](#), we take the intersection of outgoing dead symbols in predecessor nodes, instead of the union of incoming live symbols in successor nodes as in [Algorithm 7.1](#);
- On [line 13](#) of [Algorithm 7.2](#), we compute outgoing dead symbols as the union of symbols killed in the current node with incoming dead symbols, whereas [Algorithm 7.1](#) computes incoming

live symbols as the union of symbols used in the current node with non-killed outgoing live symbols;

- The dataflows is from top to bottom in [Algorithm 7.2](#), while it is from bottom to top in [Algorithm 7.1](#).

These differences underscore the role [Algorithm 7.2](#) plays as an inversion of liveness analysis.

7.4.4 Resolving Live Symbols

In many cases, it is possible to determine the set of live symbols in a cell with high precision purely via static analysis. In some cases, however, it is difficult to do so without awareness of additional runtime data. To illustrate, consider the example below:

```
x = 0 [1]
def f(y):
    return x + y
lst = [f, lambda t: t + 1]

print(lst[1](2)) [2]
```

Figure 7.6: Example of a difficult case for precise liveness checking.

Whether or not symbol `x` should be considered live at the top of the second cell depends on whether the call to `lst[1](2)` refers to the list entry containing the lambda, or the entry containing function `f`. In this case, a static analyzer might be able to infer that `lst[1]` does not reference `f` and that `x` should therefore not be considered live at the top of cell 2 (since there is no call to function `f`, in whose body `x` is live), but doing so in general is challenging due to Rice’s theorem. Instead of doing so purely statically, `NBSAFETY` performs an extra resolution step, since it can actually examine the runtime value of `lst[1]` in memory. This allows `NBSAFETY` to be more precise about live symbols than a conservative approach would be, which would be forced to consider `x` as live even though `f` is not referenced by `lst[1]`.

7.5 CELL HIGHLIGHTS

In this section, we describe how to combine the lineage metadata from [Section 7.3](#) with the output of `NBSAFETY`’s static checker to highlight cells of interest.

7.5.1 Highlight Abstraction

We begin by defining the notion of *cell highlights* in the abstract before discussing concrete examples, how they are presented, and how they are computed.

Definition 7.11 [Cell highlights]. *Given a notebook N abstractly defined as an ordered set of cells $\{c_i\}$, a set of cell highlights \mathcal{H} is a subset of N comprised of cells that are semantically related in some way at a particular point in time.*

More concretely, we will consider the following cell highlights:

- \mathcal{H}_s , the set of stale cells in a notebook;
- \mathcal{H}_f , the set of fresh cells in a notebook; and
- \mathcal{H}_r , the set of refresher cells in a notebook.

Note that these sets of cell highlights are all implicitly indexed by their containing notebook’s execution counter. When not clear from context we write $\mathcal{H}_s^{(t)}$, $\mathcal{H}_f^{(t)}$, and $\mathcal{H}_r^{(t)}$ (respectively) to make the time dependency explicit. Along these lines, we are also interested in the following “delta” cell highlights:

- $\Delta\mathcal{H}_f^{(t)} = \mathcal{H}_f^{(t)} - \mathcal{H}_f^{(t-1)}$ (new fresh cells); and
- $\Delta\mathcal{H}_r^{(t)} = \mathcal{H}_r^{(t)} - \mathcal{H}_r^{(t-1)}$ (new refresher cells)

again omitting superscripts when clear from context.

Interface. We have already seen from the example in [Figure 7.3](#) that stale cells are given staleness warnings to the left of the cell, and refresher cells are given cleanup suggestions to the left of the cell. The current version of NBSAFETY as of this writing (0.0.49) also augments fresh cells with cleanup suggestions of the same color as that used for refresher cells. Overall, the fresh and refresher highlights are intended to steer users toward cells that they may wish to re-execute, and the stale highlights are intended to steer users away from cells that they may wish to avoid, intuitions that we validate in our empirical study (§7.6). Experimenting with presentation techniques for the various sets \mathcal{H}_* is an interesting avenue that we leave to future work.

Computation. Computing \mathcal{H}_s and \mathcal{H}_f is straightforward: for each cell c , we simply run a liveness checker ([Algorithm 7.1](#)) to determine $\text{LIVE}(c)$, and then perform a metadata lookup for each symbol $s \in \text{LIVE}(c)$ to determine whether s is fresh w.r.t. c or stale. The manner in which NBSAFETY computes refresher cells deserves a more thorough treatment that we consider next.

7.5.2 Computing Refresher Cells Efficiently

Before we discuss how NBSAFETY uses the initialized variable checker from [Section 7.4](#) to efficiently compute refresher cells, consider how one might design an algorithm to compute refresher cells directly from [Definition 7.10](#). The straightforward way is to loop over all non-stale cells

Algorithm 7.3: Computing refresher cells naïvely

Input: Notebook N , stale cells $\mathcal{H}_s \subseteq N$

Output: Refresher cells \mathcal{H}_r

```
1  $\mathcal{H}_r \leftarrow \emptyset$ ;  
2 foreach  $c_s \in \mathcal{H}_s$  do  
3    $\text{STALE}(c_s) \leftarrow \text{live, stale symbols in } c_s$ ;  
4   foreach  $c_r \in N - \mathcal{H}_s$  do  
5      $\text{STALE}(c_r \oplus c_s) \leftarrow \text{live, stale symbols in } c_r \oplus c_s$ ;  
6     if  $\text{STALE}(c_s) - \text{STALE}(c_r \oplus c_s) \neq \emptyset$  then  
7        $\mathcal{H}_r \leftarrow \mathcal{H}_r \cup \{c_r\}$ ;  
8     end  
9   end  
10 end  
11 return  $\mathcal{H}_r$ ;
```

$c_r \in N - \mathcal{H}_s$ and compare whether $\text{STALE}(c_r \oplus c_s)$ is smaller than $\text{STALE}(c_s)$. In the case that \mathcal{H}_s and $N - \mathcal{H}_s$ are similar in size, this requires performing $\mathcal{O}(|N|^2)$ liveness analyses, which would create unacceptable latency in the case of large notebooks. This inefficient approach is depicted in [Algorithm 7.3](#).

By leveraging an initialized variable checker, it turns out that we can check whether $\text{STALE}(c_s)$ and $\text{DEAD}(c_r)$ have any overlap instead of performing liveness analysis over $c_r \oplus c_s$ and checking whether $\text{STALE}(c_r \oplus c_s)$ shrinks. We state this formally as follows:

Theorem 7.1. *Let N be a notebook, and let $c_s \in \mathcal{H}_s \subseteq N$. For any other $c_r \in N - \mathcal{H}_s$, the following equality holds:*

$$\text{STALE}(c_s) - \text{STALE}(c_r \oplus c_s) = \text{DEAD}(c_r) \cap \text{STALE}(c_s) \quad (7.10)$$

Proof. We show each side of the equality is a subset of the other side. First, suppose some stale symbol \mathbf{x} is live in c_s but not in $c_r \oplus c_s$. Then, at the point where control transfers from c_r to c_s in the outermost scope, every path of execution will definitely have redefined \mathbf{x} .¹ Otherwise, there would exist a path in c_r wherein \mathbf{x} is not redefined, and because \mathbf{x} is live at the top of c_s , it would also be live at the top of $c_r \oplus c_s$. As such, $\mathbf{x} \in \text{DEAD}(c_r)$ by definition. Furthermore, $\mathbf{x} \in \text{STALE}(c_s)$ by our initial assumption, so $\mathbf{x} \in \text{DEAD}(c_r) \cap \text{STALE}(c_s)$.

Conversely, suppose some stale symbol \mathbf{x} is live in c_s but dead in c_r . By definition, every path of execution in c_r redefines \mathbf{x} . We would like to say that \mathbf{x} is not live in $c_r \oplus c_s$, but deadness in c_r

¹Note that there is no way for control to transfer from c_r to c_s in the outermost scope except at the point where c_r and c_s meet lexically (technically, c_r could call a function defined in c_s , but if this were to occur, control would not be at the outermost scope).

Algorithm 7.4: Computing refresher cells efficiently

Input: Notebook N , stale cells $\mathcal{H}_s \subseteq N$,
stale and live symbols $\text{STALE}(c_s), \forall c_s \in \mathcal{H}_s$,
dead symbols $\text{DEAD}(c_r), \forall c_r \in N - \mathcal{H}_s$

Output: Refresher cells $\mathcal{H}_r \subseteq N$

```
1  $\text{DEAD}^{-1}[s] \leftarrow \emptyset, \forall s \in \text{DEAD}(c_r), \forall c_r \in N - \mathcal{H}_s;$ 
2 foreach  $c_r \in N - \mathcal{H}_s$  do
3   | foreach  $s \in \text{DEAD}(c_r)$  do
4   |   |  $\text{DEAD}^{-1}[s] \leftarrow \text{DEAD}^{-1}[s] \cup \{c_r\};$ 
5   | end
6 end
7  $\mathcal{H}_r \leftarrow \emptyset;$ 
8 foreach  $c_s \in \mathcal{H}_s$  do
9   | foreach  $s \in \text{STALE}(c_s)$  do
10  |   |  $\mathcal{H}_r \leftarrow \mathcal{H}_r \cup \text{DEAD}^{-1}[s];$ 
11  | end
12 end
13 return  $\mathcal{H}_r;$ 
```

does not preclude liveness in c_r (if, e.g., \mathbf{x} is used in some path of c_r before it is redefined later). Thus, it is only true that \mathbf{x} is not live if $c_r \oplus c_s$ if it is also not live in c_r . In fact, \mathbf{x} is not live in c_r because $c_r \notin \mathcal{H}_s$; i.e., c_r has no live stale symbols by assumption, and \mathbf{x} is stale; thus \mathbf{x} is both live in c_s and not live in $c_r \oplus c_s$; i.e., $\mathbf{x} \in \text{STALE}(c_s) - \text{STALE}(c_r \oplus c_s)$, which is what we needed to show to complete the proof.

Theorem 7.1 relies crucially on the fact that the CFG of the concatenation of two cells c_r and c_s into $c_r \oplus c_s$ will have a “choke point” at the position where control transfers from c_r into c_s , so that any symbols in $\text{DEAD}(c_r)$ cannot be “revived” in $c_r \oplus c_s$.

Computing \mathcal{H}_r Efficiently. Contrasted with taking $\mathcal{O}(|N|^2)$ pairs $c_s \in \mathcal{H}_s, c_r \in N - \mathcal{H}_s$ and checking liveness on each concatenation $c_r \oplus c_s$, **Theorem 7.1** instead allows us compute the set \mathcal{H}_r as

$$\bigcup_{c_s \in \mathcal{H}_s} \bigcup_{s \in \text{STALE}(c_s)} \{c_r \in N - \mathcal{H}_s : s \in \text{DEAD}(c_r)\} \quad (7.11)$$

Equation 7.11 can be computed efficiently according to **Algorithm 7.4**, which creates an inverted index that maps dead symbols to their containing cells (DEAD^{-1}) in order to efficiently compute the inner set union. Furthermore, **Algorithm 7.4** only requires $\mathcal{O}(|N|)$ liveness analyses and $\mathcal{O}(|N|)$ initialized variable analyses as preprocessing, translating to significant latency reductions in our benchmarks (§7.6.4).

7.6 EMPIRICAL STUDY

We now evaluate NBSAFETY’s ability to highlight unsafe cells, as well as cells that resolve safety issues (refresher cells). We do so by replaying 666 real notebook sessions and measuring how the cells highlighted by NBSAFETY correlate with real user actions. After describing data collection (§7.6.1) and our evaluation metrics (§7.6.2), we present our quantitative results (§7.6.3 and §7.6.4), followed by a qualitative comparison with other systems informed by real examples from our data (§7.6.5 and §7.6.6).

7.6.1 Notebook Session Replay Data

We now describe our data collection and session replay efforts.

Data Scraping. The `.ipynb` json format contains a static snapshot of the code present in a computational notebook and lacks explicit interaction data, such as how the code present in a cell evolves, which cells are re-executed, and the order in which cells were executed.² Fortunately, Jupyter’s IPython kernel implements a history mechanism that includes information about individual cell executions in each session, including the source code and execution counter for every cell execution. We thus scraped `history.sqlite` files from 712 repositories files using GitHub’s API [195], from which we successfully extracted 657 such files. In total, these history files contained execution logs for ≈ 51000 notebook sessions, out of which we were able to collect metrics for 666 after conducting the filter and repair steps described next.

Notebook Session Repair. Many of the notebook sessions were impossible to replay with perfect replication of the session’s original behavior (due to, e.g., missing files). To cope, we adapted ideas from Yan et al. [189] to repair sessions wherever possible. Specifically, we took the following measures:

- Since NBSAFETY runs on Python 3, we used the `2to3` tool [196] whenever we encountered Python 2 code.
- To deal with differing APIs used by different versions of the same library (e.g., `scikit-learn`), we first gathered all the import statements for each library and tried to execute them under different versions of the aforementioned library, using the version that minimized import errors to finally replay the session.
- We normalized all path-like strings to point to the same directory, to prevent invalid accesses to nonexistent directories.

²The cell counter in a `.ipynb` file only contains the latest executed cell version for each cell, and says nothing about how executions of earlier iterations of the cell are ordered w.r.t. others.

- We used the Kaggle API to search for and attempt to download any `csv` files referenced by each session.
- We removed any lines or cells that attempted to run system commands through Jupyter’s line (resp. cell) magic functionality.
- We executed the line magic `%matplotlib inline` before replaying a session to avoid rendering matplotlib charts with Qt.

Session Filtering. Despite these efforts, we were unable to reconstruct some sessions to their original fidelity due to various environment discrepancies. Furthermore, certain sessions had few cell executions and appeared to be random tinkering. We therefore filtered out sessions fitting any of the following criteria:

- Sessions with fewer than 50 cell executions;
- Sessions that attempted to run shell commands;
- Sessions that solicited user input via `readline` or other means;
- Sessions that attempted to connect to external services (e.g. AWS, Spark, Postgres, MySQL, etc.);
- Sessions that attempted to read nonexistent files (or those that could not be found using the Kaggle API).

After these steps, we were left with 2566 replayable sessions. However, we were unable to gather meaningful metrics on more than half of the sessions we replayed because of exceptions thrown upon many cell executions. We filtered these in post-processing by removing data for any session with more than 50% of cell executions resulting in exceptions.

After the repair and filtration steps, we extracted metrics from a total of 666 sessions. Our scripts are available on GitHub [197].

Environment. All experiments were conducted on a 2019 MacBook Pro with 32 GiB RAM and a Core i9 processor running macOS 10.14.5, Python 3.7, and `NBSAFETY` 0.0.49. We replayed notebook sessions in a container instance to ensure our local files would not be compromised in the event of intentionally or unintentionally malicious code present in the sessions we scraped.

7.6.2 Metrics

Besides conducting benchmark experiments to measure overhead associated with `NBSAFETY` (§7.6.4), the primary goal of our empirical study is to evaluate our system and interface design choices from the previous sections by testing two hypotheses. Our first hypothesis (i) is that *cells with staleness issues highlighted by NBSAFETY are likely to be avoided by real users*, suggesting that these cells are indeed unsafe to execute. Our second hypothesis (ii) is that *fresh and refresher cells highlighted by NBSAFETY are more likely to be selected for re-execution*, indicating that these

suggestions can help reduce cognitive overhead for users trying to choose which cells to re-execute. To test these hypotheses, we introduce the notion of *predictive power* for cell highlights.

Definition 7.12 [Predictive Power]. *Given a notebook N with a total of $|N|$ cells, the id of the next cell executed c , and a non-empty set of cell highlights \mathcal{H} (chosen before c is known), the predictive power of \mathcal{H} is defined as $\mathcal{P}(\mathcal{H}) = \mathbb{I}\{c \in \mathcal{H}\} \cdot |N|/|\mathcal{H}|$.*

Averaged over many measurements, predictive power assesses how many more times more likely a cell from some set of highlights \mathcal{H} is to be picked for re-execution, compared to random cells.

Intuition. To understand predictive power, consider a set of highlights \mathcal{H} chosen uniformly randomly without replacement from the entire set of available cells. In this case,

$$\mathbb{E}[\mathbb{I}\{c \in \mathcal{H}\}] = \mathbb{P}(c \in \mathcal{H}) = |\mathcal{H}|/|N| \quad (7.12)$$

so that the predictive power of \mathcal{H} is $(|\mathcal{H}|/|N|) \cdot (|N|/|\mathcal{H}|) = 1$. This holds for any number of cells in the set of highlights \mathcal{H} , even when $|\mathcal{H}| = |N|$. Increasing the size of \mathcal{H} increases the chance for a nonzero predictive power, but it also decreases the “payout” when $c \in \mathcal{H}$. For a fixed notebook N , the maximum possible predictive power for \mathcal{H} occurs when $\mathcal{H} = \{c\}$, in which case $\mathcal{P}(\mathcal{H}) = |N|$.

Rationale. Our goal in introducing predictive power is not to give a metric that we then attempt to optimize; rather, we merely want to see how different sets of cell highlights correlate with real user behavior. In some sense, any $\mathcal{P}(\mathcal{H}) \neq 1$ is interesting: $\mathcal{P}(\mathcal{H}) < 1$ indicates that users tend to avoid \mathcal{H} , and $\mathcal{P}(\mathcal{H}) > 1$ indicates that users tend to prefer \mathcal{H} . For the different sets of cell highlights $\{\mathcal{H}_*\}$ introduced in [Section 7.5](#), each $\mathcal{P}(\mathcal{H}_*)$ helps us to make this determination.

Gathering measurements. The session interaction data available in the scraped history files only contains the submitted cell contents for each cell execution, and unfortunately lacks cell identifiers. Therefore, we attempted to infer the cell identifier as follows: for each cell execution, if the cell contents were $\geq 80\%$ similar to a previously submitted cell (by Levenshtein similarity), we assigned the identifier of that cell; otherwise, we assigned a new identifier. Whenever we inferred that an existing cell was potentially edited and re-executed, we measured predictive power for various highlights \mathcal{H}_* when such highlights were non-empty. Across the various highlights, we computed the average of such predictive powers for each sessions, and the averaged the average predictive powers across all sessions, reporting the result as $\text{AVG}(\mathcal{P}(\mathcal{H}_*))$ for each \mathcal{H}_* (§7.6.3).

Highlights of Interest. We gathered metrics for \mathcal{H}_s , \mathcal{H}_f , $\Delta\mathcal{H}_f$, \mathcal{H}_r , and $\Delta\mathcal{H}_r$, which we described earlier in [Section 7.5](#). Additionally, we also gathered metrics for the following “baseline highlights”:

- \mathcal{H}_n , or the *next cell highlight*, which contains only the $k + 1$ cell (when applicable) if cell k was the previous cell executed; and

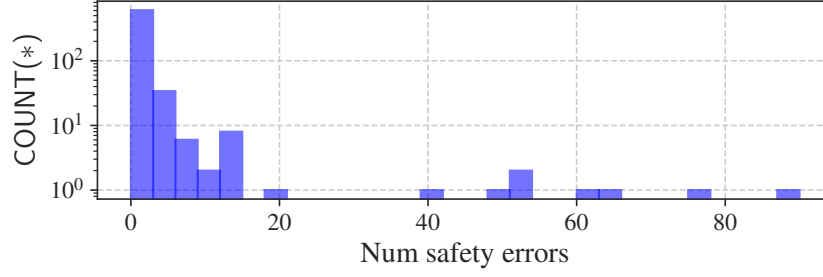


Figure 7.7: Histogram showing distribution of safety errors across sessions.

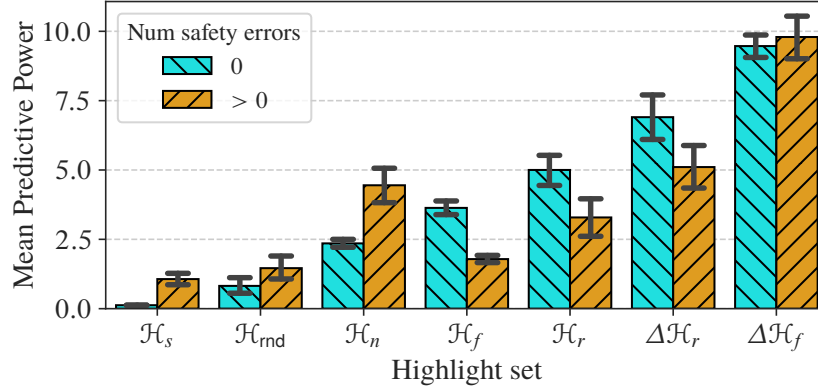


Figure 7.8: $\text{AVG}(\mathcal{P}(\mathcal{H}_*))$ for sessions with/without safety issues.

- \mathcal{H}_{rnd} , or the *random cell highlight*, which simply picks a random cell from the list of existing cells.

We take measurements for \mathcal{H}_n because picking the next cell in a notebook is a common choice, and it is interesting to see how its predictive power compares with cells highlighted by the NBSAFETY frontend such as \mathcal{H}_f and \mathcal{H}_r . We also take measurements for \mathcal{H}_{rnd} to validate via Monte Carlo simulation the claim that random cells \mathcal{H}_{rnd} should satisfy $\mathcal{P}(\mathcal{H}_{rnd}) = 1$ in expectation.

7.6.3 Predictive Power Results

In this section, we present the results of our empirical evaluation. *Overall, NBSAFETY discovered that 117 sessions out of the 666 encountered staleness issues at some point*, underscoring the need for a tool to prevent such errors. Furthermore, we found that the “positive” highlights like \mathcal{H}_f and \mathcal{H}_r correlated strongly with user choices.

Predictive Power for Various Highlights. We now discuss average $\mathcal{P}(\mathcal{H}_*)$ for the various \mathcal{H}_* we consider, summarized in [Table 7.2](#).

Quantity	\mathcal{H}_n	\mathcal{H}_{rnd}	\mathcal{H}_s	\mathcal{H}_f	\mathcal{H}_r	$\Delta\mathcal{H}_f$	$\Delta\mathcal{H}_r$
$\text{AVG}(\mathcal{P}(\mathcal{H}_*))$	2.64	1.02	0.30	2.83	3.90	9.17	6.20
$\text{AVG}(\mathcal{H}_*)$	1.00	1.00	2.71	3.73	2.31	1.73	1.81

Table 7.2: Summary of measurements taken for various highlight sets.

Summary. Out of the highlights \mathcal{H}_* with $\mathcal{P}(\mathcal{H}_*) > 1$, new fresh cells, $\Delta\mathcal{H}_f$, had the highest predictive power, while \mathcal{H}_n had the lowest (excepting \mathcal{H}_{rnd} , which had $\mathcal{P}(\mathcal{H}_{\text{rnd}}) \approx 1$ as expected). \mathcal{H}_s had the lowest predictive power coming in at $\mathcal{P}(\mathcal{H}_s) \approx 0.30$, suggesting that users do, in fact, avoid stale cells.

We measured the average value of $\mathcal{P}(\mathcal{H}_s)$ at roughly 0.30, which is the lowest mean predictive power measured out of any highlights. One way to interpret this is that users were more than $3\times$ less likely to re-execute stale cells than they are to re-execute randomly selected highlights of the same size as \mathcal{H}_s — strongly supporting the hypothesis that users tend to avoid stale cells.

On the other hand, all of the highlights \mathcal{H}_n , \mathcal{H}_f , \mathcal{H}_r , $\Delta\mathcal{H}_f$, and $\Delta\mathcal{H}_r$ satisfied $\mathcal{P}(\mathcal{H}_*) > 1$ on average, with $\mathcal{P}(\Delta\mathcal{H}_f)$ larger than the others at 9.17, suggesting that users are *more than 9× more likely* to select newly fresh cells to re-execute than they are to re-execute randomly selected highlights of the same size as $\Delta\mathcal{H}_f$. In fact, \mathcal{H}_n was the lowest non-random set of highlights with mean predictive power > 1 , strongly supporting our design decision of specifically guiding users to all the cells from \mathcal{H}_f and \mathcal{H}_r (and therefore to $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$ as well) with our aforementioned [visual cues](#). Furthermore, we found that no $|\mathcal{H}_*|$ was larger than 4 on average, suggesting that these cues are useful, and not overwhelming.

Finally, given the larger predictive powers of $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$, we plan to study interfaces that present these highlights separately from \mathcal{H}_f and \mathcal{H}_r in future work.

Effect of Safety Issues on Predictive Power. Of the 666 sessions we replayed, we detected 1 or more safety issues (due to the user executing a stale cell) in 117, while the majority (549) did not have safety issues. A histogram depicting the distribution of “# safety issues” is given in [Figure 7.7](#). We reveal interesting behavior by computing $\text{AVG}(\mathcal{P}(\mathcal{H}_*))$ when restricted to (a) sessions without safety errors, and (b) sessions with 1 or more safety errors, depicted in [Figure 7.8](#).

Summary. For sessions with safety errors, users were more likely to select the next cell (\mathcal{H}_n), and less likely to select fresh or refresher cells (\mathcal{H}_f and \mathcal{H}_r , respectively).

[Figure 7.8](#) plots $\text{AVG}(\mathcal{P}(\mathcal{H}_*))$ for various highlight sets after faceting on sessions that did and did not have safety errors. By definition, $\text{AVG}(\mathcal{P}(\mathcal{H}_s)) = 0$ for sessions without safety errors (otherwise, users would have attempted to execute one or more stale cells), but even for sessions with safety errors, we still found $\mathcal{P}(\mathcal{H}_s) < 1$ on average, though not enough to rule out random chance.

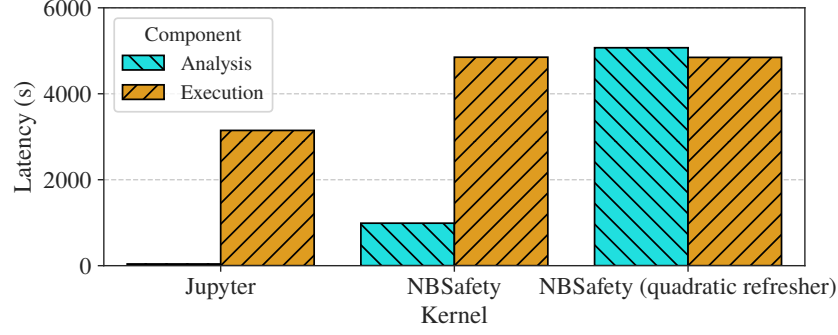


Figure 7.9: Comparing latencies of execution and analysis components for different kernels.

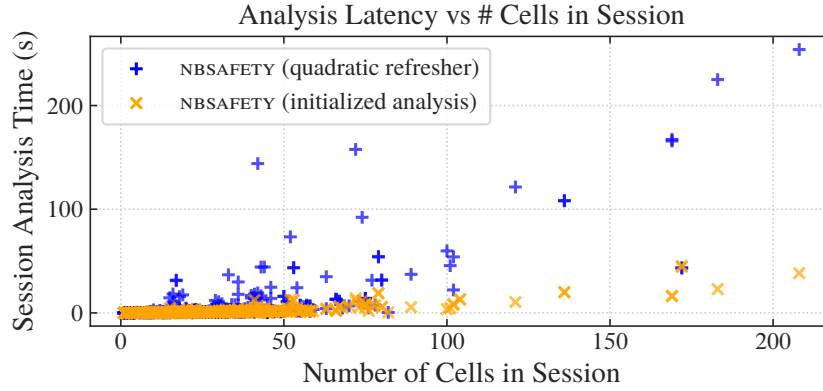


Figure 7.10: Measuring the impact of cell count on analysis latency for NBSAFETY with and without efficient refresher computation.

Interestingly, we found that $\text{AVG}(\mathcal{P}(\mathcal{H}_n))$ was significantly higher for sessions with safety issues, suggesting that users were more likely to execute the next cell without much thought.

Finally, we found that users were significantly *less* likely to choose cells from \mathcal{H}_f , \mathcal{H}_r , or $\Delta\mathcal{H}_r$ for sessions with safety errors. In fact, users favored \mathcal{H}_n over \mathcal{H}_r or \mathcal{H}_f in this case. Regardless of whether sessions had safety issues, however, $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$ still had the highest mean predictive powers out of any of the highlights, with $\text{AVG}(\mathcal{P}(\Delta\mathcal{H}_f))$ relatively unaffected by safety issues.

7.6.4 Benchmark Results

Our benchmarks are designed to assess the additional overhead incurred by our tracer and checker by measuring the end-to-end execution latency for the aforementioned 666 sessions, with and without NBSAFETY. Furthermore, we assess the impact of our initialized analysis approach to computing refresher cells (Algorithm 7.4) by comparing it with the naïve quadratic baseline (Algorithm 7.3).

Approach	Jupyter	NBSAFETY	NBSAFETY (quadratic refresher)
Total Time (s)	3150	5840	9920
Median Slowdown	1×	1.44×	1.58×

Table 7.3: Summary of latency measurements. Median slowdown measured on sessions that took > 5 seconds to execute in vanilla Jupyter.

Overall Execution Time. We summarize the time needed for various methods to replay the 666 sessions in our execution logs in [Table 7.3](#), and furthermore faceted on the static analysis and tracing / execution components in [Figure 7.9](#). We measured latencies for both vanilla Jupyter and NBSAFETY, as well as for an ablation that replaces the efficient refresher computation algorithm with the quadratic variant ([Algorithm 7.3](#)).

Summary. The additional overhead introduced by NBSAFETY is within the same order-of-magnitude as vanilla Jupyter, taking less than $2\times$ longer to replay all 666 sessions, with typical slowdowns less than $1.5\times$. Without initialized analysis for refresher computation, however, total reply time increased to more than $3\times$ the time taken by Jupyter.

Furthermore, we see from [Figure 7.9](#) that refresher computation begins to dominate with the quadratic variant, while it remains small for the linear variant based on initialized analysis.

Although NBSAFETY’s tracer introduces some additional overhead compared to the vanilla Jupyter kernel, we note that this overhead is relatively minor (less than $1.5\times$), and that a less-optimized tracing implementation would have performed far worse. For example, suppose `lst` contains one million elements, and we materialize the output of a map operation, e.g. `lst = list(lst.map(f))`. Using Python’s tracing mechanism directly, this would produce at least one million `call` and `return` events, leading to overhead in excess of $10\times$. NBSAFETY is smart enough to disable tracing if the same program statement is encountered twice during a given execution, so that this statement executes just as in vanilla Jupyter, while still detecting that symbol `lst` should be given `f` as a dependency.

Impact of Number of Cells on Analysis Latency. To better illustrate the benefit of using initialized analysis for efficient computation of refresher cells, we measured the latency of just NBSAFETY’s analysis component, and for each session, we plotted this time versus the total number of cells created in the session, in [Figure 7.10](#).

Summary. While quadratic refresher computation is acceptable for sessions with relatively few cells, we observe unacceptable per-cell latencies for larger notebooks with more than 50 or so cells.

System	Datalore	Nodebook	Dataflow	NBSAFETY
Auto infers symbol lineage	✓	✓	✗	✓
Composes with NBSAFETY	✗	✗	✓	N/A
Auto resolves staleness	✓	✓	✓*	optionally
↯always does so correctly	✓	✗	✓*	N/A‡
Preserves Jupyter semantics	✗	✗	✓†	✓
No crashing on valid Python	✗	✗	✓	✓
No other performance penalty	✗	✗	✓	✗ (minor)

Table 7.4: Summary of key distinguishing properties of notebook systems that help prevent stale executions.

* Only for manually specified dependencies.

† Except for manually specified dependencies.

‡ NBSAFETY can display the run-plan and allow manual corrections if needed.

The linear variant that leverages initialized analysis, however, scales gracefully even for the largest notebooks in our execution logs.

The variance in Figure 7.10 for notebooks of the same size can be attributed to cells with different amounts of code, as well as different numbers of cell executions (since the size of the notebook is a lower bound for the aforementioned according to our replay strategy).

7.6.5 Comparison with Other Systems

We now give a qualitative comparison of NBSAFETY with other systems that attempt to resolve staleness issues, viewed through the lens of the data we collected in our empirical study. After surveying relevant literature and open source software repositories, we are aware of three such systems: Dataflow notebooks [12], Nodebook [13], and the Datalore kernel from JetBrains [14].

The most salient distinctions for each approach are summarized in Table 7.4. We now provide a summary for each system.

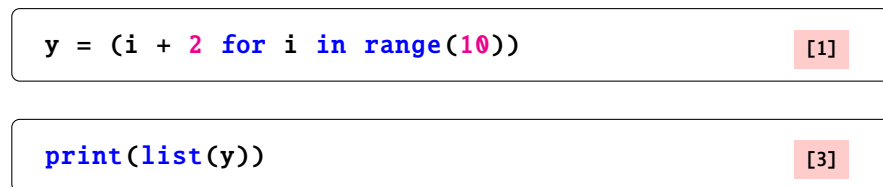
Dataflow Notebooks. While Dataflow notebooks have many desirable properties, they require the user to specify dependencies manually in order to leverage any staleness-resolving functionality. Dataflow notebooks can be used in conjunction with NBSAFETY if reactive cell execution via manually-specified dependencies is desired.

Datalore and Nodebook. Both the Datalore kernel and Nodebook seem to take a hybrid analysis / memoization approach toward automatic staleness resolution: each cell serializes the variables that it assigns, and if a cell c is rerun, a liveness checker determines what symbols need to be deserialized (using versions computed by cells that appear in c spatially) and used as “inputs” to c , possibly rerunning cells prior to c if they were edited or if they depend on an edited cell. For

example, in [Figure 7.2](#), the second cell would be automatically rerun if the user attempts to rerun the third cell after rerunning the first.

These approaches allow notebooks to emulate script-like top-to-bottom behavior, but serialization can come at significant cost for objects like large dataframes, and furthermore, not all objects are serializable, thereby rendering these approaches viable only on a much smaller set of programs, as we will see. Finally, because liveness gives a conservative overestimation of symbols used, these approaches may perform more work than necessary to rerun prior edited cells, or to deserialize possibly-needed symbols.

Ability to run valid Python. Perhaps the most serious shortcoming of memoization-based approaches stems from their failure to execute valid Python code. Consider the following example:



```
y = (i + 2 for i in range(10)) [1]
```

```
print(list(y)) [3]
```

Figure 7.11: The first cell contains a generator object, which does not serialize in Python.

If the user edits the first cell and then attempts to run the second cell twice using either Nodebook or the Datalore kernel, they will observe an error when these approaches try (and fail) to load the non-serializable object `y` from storage.

Ability to conduct multiverse analyses. To facilitate the below discussion, we define the “rerun all cells” approach adopted by Nodebook, Datalore, and Dataflow notebooks (for manual dependencies) to be a “forcible cascade” approach, the selective rerun approach adopted by NBSAFETY to be a “supervised permissive cascade” approach, and that of Jupyter to be a “manual cascade” approach.

In exploratory programming and data analysis, users do not usually have a clear indication of which approach might work well up-front [198]. So, they typically try various alternative approaches to achieve their end-goal, while also recording snippets of what they had tried previously, for reuse, and for returning to old alternatives [199]. The forcible cascade approach in this case has the unintended effect of having all of their downstream alternatives being executed, when the user wanted to execute precisely one.

These sorts of multiverse analyses are hindered by the forcible cascade approaches. In fact, we found several instances in our execution logs wherein users explicitly saved off variables to be returned to later, and where forcible cascades would have overwritten these variables’ saved values. We now give a typical example, depicted below:

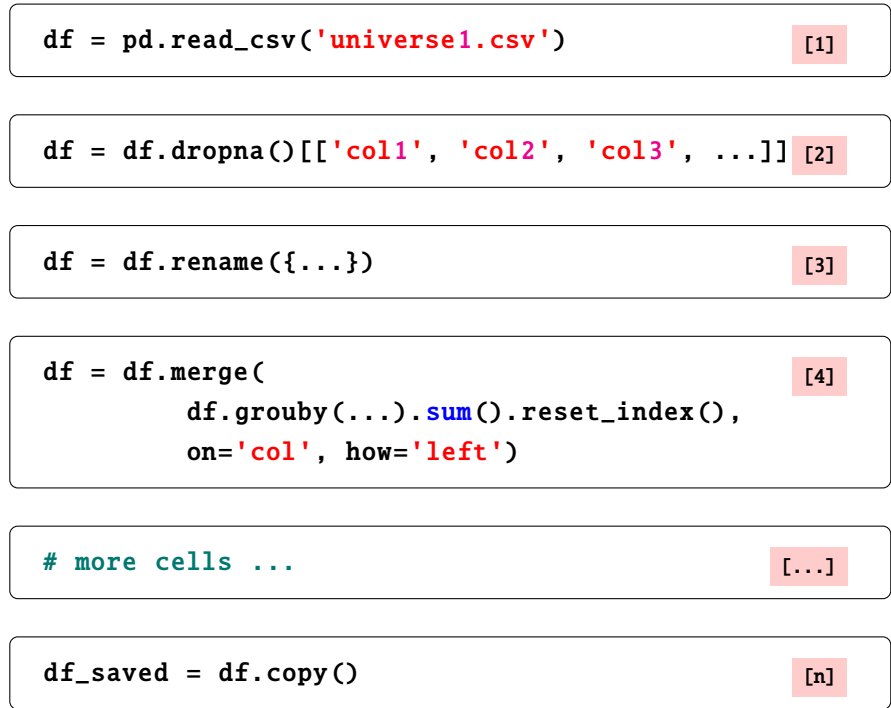


Figure 7.12: Example notebook session performing a multiverse analysis.

After reading the file `universe1.csv` into a dataframe and performing some transformations, the user would then save a copy of the transformed dataframe. The user would then repeat the same transformations by changing cell 1 to read in `universe2.csv` and manually running the cells below, but stopping before creating the copy. The user then would perform some comparison between the transformed `universe1.csv` data and the transformed `universe2.csv` data. Note that a forcible cascade would have overwritten the variable `df_saved`, preventing this comparison.

7.6.6 Staleness Case Study

We now discuss a particularly egregious example of unsafe behavior in one of the 666 replayed sessions that would have been caught by `NBSAFETY`. In this session, the user was attempting to visualize a Wiener process defined by the following function:

```
def wiener(tmax, n):  
    # Return one realization of a Wiener process  
    # with n steps and a max time of tmax.  
    times = np.linspace(0, tmax, n)  
    difference = np.diff(times)  
    process = np.random.normal(0, difference**.5)  
    process = np.cumsum(process)  
    return times, process
```

Figure 7.13: Staleness case study (step 1).

The user then initially called this function and saved the output in variables `t` and `w`:

```
t, w = wiener(1.0, 1000)
```

Figure 7.14: Staleness case study (step 2).

After inspecting a few values in the array `w`, the user then decided to rename it from lowercase `w` to uppercase `W`:

```
t, W = wiener(1.0, 1000)
```

Figure 7.15: Staleness case study (step 3).

Next, the user used the popular visualization library Altair [\[200\]](#) to plot the output of the `wiener` function, using the following code (and producing output similar to the figure below the cell):


```
data = pd.DataFrame({'time': t, 'W': w})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```

[4]

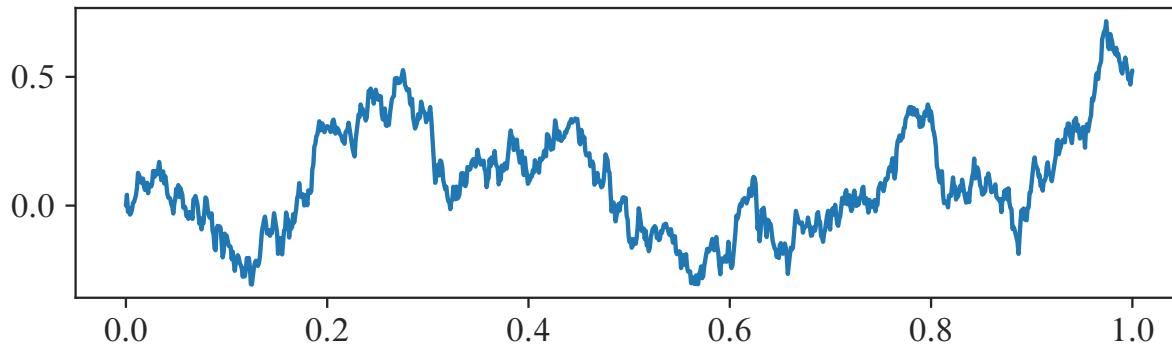


Figure 7.16: Staleness case study (step 4).

However, note that in cell 4, the dataframe created by the user, `data`, refers to the old lowercase `w`, and not the new uppercase `W` most recently created. Thus, when the user reran cells 3 and 4 in succession, the exact same figure as that from the original cell 4 was generated.

Confused, the user then reran cells 1, 3, and 4 in succession, each time generating a plot identical to that from the original cell 4. This process repeated itself around 20 times, before the user finally noticed the problem and changed cell 4 to the following:

```
data = pd.DataFrame({'time': t, 'W': W})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```

[4]

Figure 7.17: Staleness case study (last step).

This at last generated a different plot from the output of the original cell 4, but the entire process resulted wasted effort that would have been saved had the user noticed the error earlier.

How NBSAFETY Helps. To see how NBSAFETY would have helped, let us examine the highlights that NBSAFETY would have presented after the user reran 3 and 4, got confused, and then reran cell 1. The state of the notebook would have then appeared similar to the following:

```
def wiener(tmax, n):
    # Return one realization of a Wiener process
    # with n steps and a max time of tmax.
    times = np.linspace(0, tmax, n)
    difference = np.diff(times)
    process = np.random.normal(0, difference**.5)
    process = np.cumsum(process)
    return (times, np.insert(process, 0, 0))
```

```
t, W = wiener(1.0, 1000)
```

```
data = pd.DataFrame({'time': t, 'W': w})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```

Figure 7.18: Highlights displayed by NBSAFETY for staleness case study after rerunning cells 3, 4, and 1 (Figures 7.13, 7.15 and 7.16, respectively) in order.

That is, cell 3 would be given a [fresh](#) cleanup suggestion highlight, because the `wiener` symbol was recently updated when the user reran the first cell (now labeled as 7). Likewise, cell 6 is given a [stale](#) highlight because lowercase `w` depends on the old version of `wiener`.

Next, when the user reruns the cell labeled as 5 in the above notebook, they would expect the unsafe highlight over cell 6 to be replaced with a fresh highlight, because they refreshed the symbol `W`. However, this does not occur, and the unsafe highlight remains. The user could have then query NBSAFETY's API to determine why, and would have been presented with the following:

```
t, W = wiener(1.0, 1000) [7]
```

```
data = pd.DataFrame({'time': t, 'W': w}) [6]
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
# WARNING: `w` (latest update in cell 2) may depend
#         on old version of symbol(s) [`wiener`]`.
```

Figure 7.19: Highlights displayed by NBSAFETY after rerunning cell 5 in Figure 7.18.

At this point, the user likely would have noticed that cell 6 does not refer to symbol W , which was updated when the user ran cell 7, but on symbol w , which lingers in notebook state from when the user originally ran cell 2.

7.7 SUMMARY

We presented NBSAFETY, a kernel and frontend for Jupyter that attempts to detect and correct potentially unsafe interactions in notebooks, all while preserving the flexibility of familiar any-order notebook semantics. We described the implementation of NBSAFETY’s tracer, checker, and frontend, and how they integrate into existing notebook workflows to *efficiently* reduce error-proneness in notebooks. We showed how cells that NBSAFETY would have warned as unsafe were actively avoided, and cells that would have been suggested for re-execution were prioritized by real users on a corpus of 666 real notebook sessions.

CHAPTER 8: CONCLUSION AND FUTURE WORK

Data science is highly *interactive*, with the analysis and development process in many ways determined adaptively while browsing, plotting visualizations, generating reports, and modeling data. To enable interactivity when working with large datasets, data scientists must typically resort to shortcuts to help cope with the scale of the data. Prior work facilitating these shortcuts would typically involve sacrificing safety for interactivity, or would otherwise completely fail to scale to modern workloads due to safety constraints. In this dissertation, we showed how to leverage distributional context to enable safe interaction at scale for a set of common data science tasks, namely *ML model development*, *browsing*, *visual exploration*, *report generation*, and *interactive notebook development*.

8.1 KEY TAKEAWAYS AND DESIGN PRINCIPLES

Our experience suggests two key takeaways and design principles when designing safe and interactive systems for data science tasks and environments.

Design Principle 1: Embrace Imperfection. Working with data is an inherently messy, iterative, and incremental process. Achieving perfect safety and interactivity simultaneously is, at the very least, challenging, if not impossible in most cases. In this dissertation, we have seen how, by judiciously accepting bounded reductions in one trait (i.e., safety or interactivity), we can greatly improve upon the other trait. We now give some specific examples:

In Chapter 4 (on index structures for browsing), we saw how accepting false positives (by leveraging Bloom index lookups) enabled design decisions that resulted in enormous space savings when indexing multidimensional data.

In Chapters 5 and 6 (on visualization and report generation), we saw how accepting some amount of (bounded) error enabled interactivity improvements in the form of order-of-magnitude speedups due to early stopping.

In Chapter 7 (on computational notebooks), we saw how, by accepting some amount of lineage overhead (bounded by the amount of code written by the data scientist), we were able to provide useful suggestions for what cells to execute and what cells to avoid due to staleness issues.

Design Principle 2: Context is Key. By relaxing either safety or interactivity in a bounded way, Design Principle 1 suggests that we can make disproportionate improvements to the other aspect. However, our experience suggests that, in order to do so, we must understand properties of the data or workload in question that enable these improvements. As before, we now draw upon specific examples from each chapter:

In *Chapter 3* (on iterative ML development), we saw how we could leverage reuse across iterations during the machine learning development lifecycle to vastly improve training and evaluation latencies, and thereby interactivity. However, such reuse was only made possible thanks to similarities across iterations — a key form of workload context.

In *Chapter 4*, our multidimensional learned Bloom index space savings required some amount of workload-aware knowledge, in the form of a negative query distribution. It is only when this negative query distribution is significantly different from the set of in-index elements that such compression is possible.

In *Chapters 5 and 6*, we were able to develop approximation techniques for interactive analytics that yielded significant speedups over existing with-guarantees techniques, but such speedups were largely on account of data characteristics. In particular, we were able to achieve earlier stopping without sacrificing correctness guarantees precisely because the data over which we were operating were “better-behaved” compared to worst-case data (thanks to lower intra- or inter-group variance).

8.2 FUTURE WORK

Following the promising results demonstrated in this dissertation, we now outline a few possible future directions.

1. Safe and Interactive Spreadsheets. Recent work [201, 202] has made great strides toward improving spreadsheet interactivity; however, comparatively little attention has been given to the safety aspect. The reference-based computation of spreadsheets can be brittle: consider a column wherein each entry refers to the previous. If some entry is deleted (e.g., due to a cut and paste interaction), it can render the results of all subsequent entries meaningless, and can be difficult to catch. How can we make spreadsheet references more robust?

2. Safe and Interactive Data Labeling. Automated labeling techniques are increasingly important for alleviating the significant manual burden associated with curating high-quality training data. One approach allows data scientists to write “labeling functions” that are then de-noised in a post-processing step [203]. In practice, the development and curation of labeling functions is itself an iterative and interactive process, but unlike in the HELIX setting, it is typically the data that change across iterations (due to changes in labeling functions) as opposed to model changes. Although significant portions of the generated training labels may be identical, leveraging reuse in this case, as in the HELIX setting, is error-prone. Can we develop a system that permits reuse of training labels across iterations in a safe manner? If so, under what modeling scenarios?

3. Improving State Management in Computational Notebooks. In [Chapter 7](#), we explored how to guard against one type of common unsafe interaction in computational notebooks (namely, stale executions), but notebooks admit many state management challenges that affect safety. The following directions immediately present themselves:

i. Categorizing and mitigating other unsafe interactions. Notebooks admit many other ways to make mistakes than just stale executions, such as typos in variable names that reference zombie values not visible in the notebook, but still present in memory. Such mistakes lead to false insights and hinder reproducibility, so understanding and mitigating against a broader spectrum of errors is extremely valuable.

ii. Enforcing atomicity and idempotence of cell executions. Atomicity and idempotence have long been known by the database community to be desirable properties of mutating interactions, and it is thus natural to ask whether we can further improve safety of computational notebooks by enforcing such semantics in the context of cell executions. Notebook cells bear a remarkable resemblance to transactions, each of which update the notebook’s internal state upon execution.

While enforcing full atomicity (e.g., by using software transactional memory [\[204\]](#)) may result in unacceptable performance, we may be able to use program analysis techniques to provide warnings when a cell would throw an exception before finishing execution, e.g., due to a type error, or undefined variable reference. And while it may be impossible to catch all such errors, we may be able to provide undo / redo functionality in some limited cases to partially mitigate the issue, similar to a transaction rollback.

Likewise, enforcing full idempotent semantics may be too heavyweight, but we may be able to use tracing to warn when re-executing a cell would likely yield a different result, if, e.g., we detect that the cell writes to the notebook state after reading a value that was last written by that same cell.

Our goal in outlining the directions above is to be representative, but not exhaustive. Overall, the techniques described in this dissertation serve as a promising framework for enabling safe interactivity while addressing scaling challenges in modern, interactive data science.

APPENDIX A: PROOFS OF SELECTED THEOREMS

A.1 PROOF OF THEOREM 3.2

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

Integer Linear Programming Formulation. Problem 3.1 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:

$$X_{a_i} = \mathbb{I} \{s(n_i) \neq S_p\} \quad (\text{A.1})$$

$$X_{b_i} = \mathbb{I} \{s(n_i) = S_c\} \quad (\text{A.2})$$

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:

$$\begin{aligned} & \underset{X_{a_i}, X_{b_i}}{\text{minimize}} && \sum_{i=1}^{|N|} X_{a_i} l_i + X_{b_i} (c_i - l_i) \end{aligned} \quad (\text{A.3a})$$

$$\text{subject to} \quad X_{a_i} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \quad (\text{A.3b})$$

$$\sum_{n_j \in \text{Pa}(n_i)} X_{a_j} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \quad (\text{A.3c})$$

$$X_{a_i}, X_{b_i} \in \{0, 1\}, \quad 1 \leq i \leq |N| \quad (\text{A.3d})$$

Equation (A.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. Equation (A.3c) is equivalent to Constraint 3.2 — 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. Equation (A.3d) requires the solution to be integers.

This formulation does not specify a constraint corresponding to Constraint 3.1. Instead, we enforce Constraint 3.1 by setting the load and compute costs of nodes that need to be recomputed to specific values, as inputs to Problem 3.1. 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 Equation (A.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 Equation (A.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 [205]. In fact, it turns out that the above problem is the dual of a flow problem; specifically, it is a minimum cut problem [206, 207]. This motivates the reduction introduced in Section 3.5.2.

Main proof. The proof for Theorem 3.2 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:

$$s(n_i) = \begin{cases} S_c & \text{if } a_i \in A \text{ and } b_i \in A \\ S_l & \text{if } a_i \in A \text{ and } b_i \notin A \\ S_p & \text{if } a_i \notin A \text{ and } b_i \notin A \end{cases} \quad (\text{A.4})$$

Lemma A.1. *A feasible solution to PSP under φ also produces a feasible solution to OEP.*

Proof. We first show that satisfying the prerequisite constraint in PSP leads to satisfying Constraint 3.2 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 Equation (A.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 A.2. *An optimal solution to PSP is also an optimal solution to OEP under φ .*

Proof. 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 objective for PSP can then be written as

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} Y_{a_i} p(a_i) + Y_{b_i} p(b_i) \quad (\text{A.5})$$

Substituting our choice for $p(a_i)$ and $P(b_i)$, Equation (A.5) becomes

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} -Y_{a_i} l_i + Y_{b_i} (l_i - c_i) \quad (\text{A.6})$$

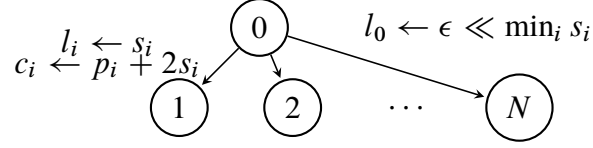


Figure A.1: OMP DAG for Knapsack reduction.

$$= \max_{Y_{a_i}, Y_{b_i}} - \sum_{i=1}^{|N|} (Y_{a_i} - Y_{b_i}) l_i + Y_{b_i} c_i \quad (\text{A.7})$$

The mapping established by Equation (A.4) is equivalent to setting $X_{a_i} = Y_{a_i}$ and $X_{b_i} = Y_{b_i}$. Thus the maximization problem in Equation (A.7) is equivalent to the minimization problem in Equation (A.3a), and we obtain an optimal solution to OEP from the optimal solution to PSP.

A.2 PROOF OF THEOREM 3.3

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:

$$n_i^t \equiv n_i^{t+1} \quad \forall n_i^t \in N_t, n_i^{t+1} \in N_{t+1} \quad (\text{A.8})$$

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 Equation (A.8) uses a reduction from the known NP-hard Knapsack problem.

Problem A.1. (*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 $S^* =$

$$\operatorname{argmax}_{S \subseteq N} \sum_{i \in S} p_i \quad (\text{A.9})$$

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 A.1. 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 Equation (3.4) 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 A.1. *We obtain an optimal solution to the Knapsack problem for $X_i = Y_i \quad \forall i \in \{1, 2, \dots, n\}$.*

Proof. 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 Equation (3.4) as

$$\operatorname{argmin}_{\mathbf{Y} \in \{0, 1\}^N} \sum_{i=1}^N Y_i l_i + \left(\sum_{i=1}^N Y_i l_i + (1 - Y_i) c_i \right) \quad (\text{A.10})$$

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 (A.10), we obtain $\operatorname{argmin}_{\mathbf{Y} \in \{0, 1\}^N} \sum_{i=1}^N -Y_i p_i$. Clearly, satisfying the storage constraint also satisfies the budget constraint in Knapsack by construction. Thus, the optimal solution to OMP as constructed gives the optimal solution to Knapsack.

A.3 PROOF OF THEOREM 4.1

We now give a proof of Theorem 4.1, which states that, the optimal classifier threshold for a sandwiched learned Bloom filter is the point that maximizes the KL divergence between the true and false positive rates $\text{KL}(X_{T_p} || X_{F_p})$, when the optimal filter has a nonempty prefilter.

Proof. Following the analysis in [64], the overall false positive rate of a sandwiched filter with b bits per key and a learned model with false positive rate F_p and false negative rate F_n is given by

$$\alpha^{b-b_2} \left(F_p + (1 - F_p) \alpha^{b_2/F_n} \right) \quad (\text{A.11})$$

where the second filter has b_2 bits per key (so that the first filter has $b - b_2$ bits per key), and $\alpha \approx 0.6185$ is chosen assuming each traditional filter uses an optimal number of hash functions (in the sense of minimizing their respective false positive rates). In more detail, the first filter has a false positive rate of α^{b-b_2} and the second filter has a false positive rate of α^{b_2/F_n} , where the “effective bits per key” b_2/F_n in the second traditional filter is scaled up because only the model false negatives need be inserted.

Thus, the overall bits per key in an optimal sandwiched filter is given by

$$b = \log_{\alpha}(FPR) + b_2 - \log_{\alpha} \left(F_p + (1 - F_p)\alpha^{b_2/F_n} \right) \quad (\text{A.12})$$

where FPR gives the desired false positive rate of the whole sandwiched setup. Mitzenmacher showed in [64] that the optimal value of b_2 is given by

$$b_2 = F_n \log_{\alpha} \left(\frac{F_p F_n}{(1 - F_p)(1 - F_n)} \right) \quad (\text{A.13})$$

independently of the overall bits per key b . Therefore, we can attempt to choose a classifier cutoff which leads to model false positive and false negative rates F_p and F_n that minimize the overall bits per key, equivalent to minimizing

$$b_2 - \log_{\alpha} \left(F_p + (1 - F_p)\alpha^{b_2/F_n} \right) \quad (\text{A.14})$$

independently of the target false positive rate FPR . Plugging in the optimal value for b_2 given above, we are minimizing

$$b_2 - \log_{\alpha} \left(F_p + (1 - F_p)\alpha^{b_2/F_n} \right) \quad (\text{A.15})$$

$$= F_n \log_{\alpha} \left(\frac{F_p F_n}{(1 - F_p)(1 - F_n)} \right) - \log_{\alpha} \frac{F_p}{1 - F_n} \quad (\text{A.16})$$

$$= F_n \log_{\alpha} \frac{F_n}{1 - F_p} + F_n \log_{\alpha} \frac{F_p}{1 - F_n} - \log_{\alpha} \frac{F_p}{1 - F_n} \quad (\text{A.17})$$

$$= F_n \log_{\alpha} \frac{F_n}{1 - F_p} + (1 - F_n) \log_{\alpha} \frac{1 - F_n}{F_p} \quad (\text{A.18})$$

$$= \frac{1}{\log \alpha} \text{KL}(F_n || 1 - F_p) \quad (\text{A.19})$$

$$= \frac{1}{\log \alpha} \text{KL}(1 - F_n || F_p) \quad (\text{A.20})$$

$$= \frac{1}{\log \alpha} \text{KL}(T_p || F_p) \quad (\text{A.21})$$

where $T_p = 1 - F_n$ is the model's true positive rate. Because $\log \alpha < 0$, our final result is that, in order to minimize the bits per key b of the sandwiched filter, we should choose the model's classification threshold in order to maximize the KL divergence between the Bernoulli distributions of true positives and false positives.

A.4 PROOF OF THEOREM 5.1

We now give a proof of [Theorem 5.1](#), which we restate for convenience.

Theorem 5.1. *Suppose we have taken n_i samples with replacement for some candidate i 's histogram, resulting in the empirical estimate \mathbf{r}_i . Then \mathbf{r}_i has ε_i -deviation with probability greater than $1 - \delta_i$ for $\varepsilon_i = \sqrt{\frac{2}{n_i} \left(|V_X| \log 2 + \log \frac{1}{\delta_i} \right)}$. That is, with probability $> 1 - \delta_i$, we have: $\|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 < \varepsilon_i$.*

Proof. For $j \in [|V_X|]$, we use r_j to denote the number of occurrences of attribute value j among the n_i samples, and the normalized count \bar{r}_j is our estimate of \bar{r}_j^* , the true proportion of tuples having value j for attribute X . Note that we have omitted the candidate subscript i for clarity.

We need to introduce some machinery. Consider functions of the form $f : [|V_X|] \rightarrow \{+1, -1\}$. Let $\{f_m\}$ be the set of all such functions, where $m \in [2^{|V_X|}]$, since there are $2^{|V_X|}$ such functions. For any $m \in [2^{|V_X|}]$, consider the random variable

$$Y_m = \sum_{j=1}^{|V_X|} f_m(j)(\bar{r}_j - \bar{r}_j^*) \quad (\text{A.22})$$

By linearity of expectation, it's clear that $\mathbb{E}[Y_m] = 0$, since $f_m(j)$ is constant and $\mathbb{E}[\bar{r}_j] = \bar{r}_j^*$ for each j . Since each \bar{r}_j is a function of the samples taken $\{s_k : 1 \leq k \leq n_i\}$, each Y_m is likewise uniquely determined from samples, so we can write $Y_m = g_m(s_1, \dots, s_{n_i})$, where each sample s_k is a random variable distributed according to $s_k \sim \bar{\mathbf{r}}^*$. Note that the function g_m satisfies the Lipschitz property

$$|g_m(s_1, \dots, s_k, \dots, s_{n_i}) - g_m(s_1, \dots, s'_k, \dots, s_{n_i})| \leq \frac{2}{n_i} \quad (\text{A.23})$$

for any $j \in [|V_X|]$ and s_1, \dots, s_{n_i} . For example, this will occur with equality if $f_m(s_k) = -f_m(s'_k)$; that is, if f_m assigns opposite signs to s_k and s'_k , then changing this single sample moves $1/n_i$ of the empirical mass in such a way that it does not get canceled out. We may therefore apply the method of bounded differences [\[148\]](#) to yield the following McDiarmid inequality—a generalization of the standard Hoeffding's inequality [\[155\]](#):

$$\mathbb{P}(Y_m \geq \mathbb{E}[Y_m] + \varepsilon_i) \leq \exp(-\varepsilon_i^2 n_i / 2) \quad (\text{A.24})$$

Recalling that $\mathbb{E}[Y_m] = 0$, this actually says that

$$\mathbb{P}(Y_m \geq \varepsilon_i) \leq \exp(-\varepsilon_i^2 n_i / 2) \quad (\text{A.25})$$

This holds for any $m \in [2^{|V_X|}]$. Union bounding over all such m , we have that

$$\mathbb{P}(\exists m : Y_m \geq \varepsilon_i) \leq 2^{|V_X|} \exp(-\varepsilon_i^2 n_i / 2) \quad (\text{A.26})$$

If this does not happen (i.e., for every Y_m , we have $Y_m < \varepsilon_i$), then we have that $\|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 < \varepsilon_i$, since for any attribute value j , $|\bar{r}_j - \bar{r}_j^*| = \max_{t_j \in \{+1, -1\}} t_j (\bar{r}_j - \bar{r}_j^*)$. But if $Y_m < \varepsilon_i$ for all m , this means that we must have some m such that

$$\varepsilon_i > \sum_j f_m(j)(\bar{r}_j - \bar{r}_j^*) = \sum_j |\bar{r}_j - \bar{r}_j^*| = \|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 \quad (\text{A.27})$$

As such $\mathbb{P}(\exists m : Y_m \geq \varepsilon_i)$ is an upper bound on $\mathbb{P}(\|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_i^*\|_1 \geq \varepsilon_i)$. The desired result follows from noting that

$$\delta_i \leq 2^{|V_X|} \exp(-\varepsilon_i^2 n_i / 2) \quad (\text{A.28})$$

$$\iff \varepsilon_i \leq \sqrt{\frac{2}{n_i} \left(|V_X| \log 2 + \log \frac{1}{\delta_i} \right)} \quad (\text{A.29})$$

A.5 PROOF OF THEOREM 6.1

In Section 6.2.2, we claimed that the DKW inequality holds for sampling without replacement from a finite population; we now sketch the proof.

Theorem 6.1. *For any $N > 0$, the DKW inequality applies for sampling without replacement from a finite dataset of size N .*

Proof. Sketch: following the original paper from Wald and Wolfowitz on confidence limits for CDFs [208], it suffices to consider the CDF for mass distributed uniformly at each integer $1, 2, \dots, N$. For each without-replacement sample size m and each deviation ε , we would like to be able to claim that

$$\mathbb{P}\left(\sup |F_N - \hat{F}_{N,m}| \geq \varepsilon\right) < \mathbb{P}\left(\sup |F_{N'} - \hat{F}_{N',m}| \geq \varepsilon\right) \quad (\text{A.30})$$

for every $N' > N$ — that is, in some sense, the CDF becomes monotonically “harder” to estimate as we increase the dataset size. Unfortunately, this turns out to not be the case, but in fact the claim follows if we merely prove the weaker condition that

$$\mathbb{P}\left(\sup |F_N - \hat{F}_{N,m}| \geq \varepsilon\right) < \mathbb{P}\left(\sup |F_{N'} - \hat{F}_{N',m}| \geq \varepsilon\right) \quad (\text{A.31})$$

for *infinitely many* $N' > N$, implying that, as $N' \rightarrow \infty$, the resulting probability to which $\mathbb{P}\left(\sup |F_{N'} - \widehat{F}_{N',m}|\right)$ converges (necessarily bounded by the probability computed in the DKW inequality) is an upper bound for the corresponding probability at every finite N' , from which the claim would follow.

We show this via construction: namely, we show that, for every N , m , and ε ,

$$\mathbb{P}\left(\sup |F_N - \widehat{F}_{N,m}| \geq \varepsilon\right) < \mathbb{P}\left(\sup |F_{2N} - \widehat{F}_{2N,m}| \geq \varepsilon\right) \quad (\text{A.32})$$

That is, the CDF becomes monotonically harder to estimate each time we double the dataset size. To show this, we consider two cases. Case 1: if point $2i - 1$ is sampled, then point $2i$ is not sampled, and vice versa for every $i = 1, \dots, N$. Conditioned on this event, the probability that $\sup |F_{2N} - \widehat{F}_{2N,m}| \geq \varepsilon$ at least the (unconditioned) probability that $\sup |F_N - \widehat{F}_{N,m}| \geq \varepsilon$, since samples at odd indices can only increase the deviation, and samples at even indices cannot decrease it. Case 2: there is at least one i for which points at both indices $2i - 1$ and $2i$ are sampled. Each such point is conceptually similar to reducing m by 1 in the original dataset of size N , but randomly weighting one of the samples by 2 instead of 1. It can be shown that each time this is done, the probability that $\sup |F_N - \widehat{F}_{N,m}| \geq \varepsilon$ increases.

A.6 PROOF OF THEOREM 6.2

In this section, we prove [Theorem 6.2](#), which states that the bounds computed by our range trimming technique (see [Algorithm 6.4](#)) fail to enclose the average of some data of interest $\text{AVG}(\mathcal{D})$ with probability less than δ . For the sake of simplicity, our analysis assumes that \mathcal{D} contains no duplicate values, although we show how to remove this assumption at the end of this section. We begin by proving a crucial lemma about the sampling distribution of $S - \{\max S\}$, given that S was sampled uniformly without-replacement from \mathcal{D} .

Lemma A.3. *Given a dataset \mathcal{D} of N unique real values in $[a, b]$ and a uniform without-replacement sample S of m values from \mathcal{D} , if we denote $b' = \max S$, the set $S - \{b'\}$ takes the distribution of a uniform without-replacement sample from $\mathcal{D}_{<b'} = \mathcal{D} \cap [a, b')$, for any applicable value of $b' \in \mathcal{D}$.*

Proof. Because S is drawn uniformly without-replacement from \mathcal{D} , any particular instance satisfies

$$\mathbb{P}_{\mathcal{D}}[S = s] = \left(\frac{|\mathcal{D}|}{|s|}\right)^{-1} \mathbb{I}\{s \subseteq \mathcal{D}\} = \left(\frac{N}{m}\right)^{-1} \mathbb{I}\{s \subseteq \mathcal{D}\} \quad (\text{A.33})$$

where we use the notation $\mathbb{P}_{\mathcal{D}}[S = s]$ to denote the probability that s was drawn uniformly without-replacement from \mathcal{D} , and $\mathbb{I}\{\cdot\}$ denotes the indicator function. We need to show that, for any $b' \in \mathcal{D}$,

$$\mathbb{P}_{\mathcal{D}}[S = s | \max S = b'] = \mathbb{P}_{\mathcal{D}_{<b'}}[S = s - \{b'\}] \mathbb{I}\{\max(s) = b'\} \quad (\text{A.34})$$

First, letting s' be any set such that $|s'| = m - 1$, we have that

$$\mathbb{P}_{\mathcal{D}_{<b'}}[S = s'] = \binom{|\mathcal{D}_{<b'}|}{m-1}^{-1} \mathbb{I}\{s' \subseteq \mathcal{D}_{<b'}\} \quad (\text{A.35})$$

Next, consider $\mathbb{P}_{\mathcal{D}}[S = s | \max S = b']$. Bayes' rule gives that

$$\mathbb{P}_{\mathcal{D}}[S = s | \max S = b'] = \frac{\mathbb{P}_{\mathcal{D}}[S = s \wedge \max S = b']}{\mathbb{P}_{\mathcal{D}}[\max S = b']} \quad (\text{A.36})$$

We have $\mathbb{P}_{\mathcal{D}}[S = s \wedge \max S = b'] = \mathbb{P}_{\mathcal{D}}[S = s] \mathbb{I}\{\max(s) = b'\}$ which is a known quantity, so the key is to compute the denominator $\mathbb{P}_{\mathcal{D}}[\max S = b']$. Using the assumption that \mathcal{D} contains unique elements, we may proceed by analogy with binary strings. The rank of b' within \mathcal{D} (starting from the smallest element) is $1 + |\mathcal{D}_{<b'}|$, so we need to compute the number of binary strings of length N containing m 1's and $(N - m)$ 0's such that position $1 + |\mathcal{D}_{<b'}|$ has a 1, and the remaining $(m - 1)$ 1's are all at positions less than $1 + |\mathcal{D}_{<b'}|$. This is precisely the same as the number of binary strings of length $|\mathcal{D}_{<b'}|$ with $(m - 1)$ 1's and $(|\mathcal{D}_{<b'}| - m + 1)$ 0's. Putting everything together,

$$\mathbb{P}_{\mathcal{D}}[S = s | \max S = b'] = \frac{\mathbb{P}_{\mathcal{D}}[S = s] \mathbb{I}\{\max(s) = b'\}}{\mathbb{P}_{\mathcal{D}}[\max S = b']} \quad (\text{A.37})$$

$$= \frac{\binom{N}{m}^{-1} \mathbb{I}\{s \subseteq \mathcal{D} \wedge \max(s) = b'\}}{\binom{|\mathcal{D}_{<b'}|}{m-1} / \binom{N}{m}} \quad (\text{A.38})$$

$$= \binom{|\mathcal{D}_{<b'}|}{m-1}^{-1} \mathbb{I}\{s \subseteq \mathcal{D} \wedge \max(s) = b'\} \quad (\text{A.39})$$

$$= \binom{|\mathcal{D}_{<b'}|}{m-1}^{-1} \mathbb{I}\{s - \{b'\} \subseteq \mathcal{D}_{<b'} \wedge \max(s) = b'\} \quad (\text{A.40})$$

$$= \mathbb{P}_{\mathcal{D}_{<b'}}[S = s - \{b'\}] \mathbb{I}\{\max(s) = b'\} \quad (\text{A.41})$$

which is precisely what we wanted to show.

Wrinkle in Lemma A.3 and Fix. The proof of Lemma A.3 assumes unique values; we show here how to remove this assumption without loss of generality. The uniqueness assumption as used is necessary only to ensure that elements of \mathcal{D} are *totally ordered* under some relation “ $<$ ” (with “ $<$ ” \equiv “ $<$ ” in the proof). To fix, we can simply augment every $v \in \mathcal{D}$ with an additional unique *label*

(where the set of labels are totally ordered) such that item v becomes $v' \equiv (v, v_i)$. Then, define “ \prec ” as a relation such that $v' \prec w'$ if $v < w$, or $v = w$ and $v_i < w_i$. In this way, any $v', w' \in \mathcal{D}'$ satisfy exactly one of $v' \prec w'$ or $w' \prec v'$, and the proof of [Lemma A.3](#) goes through, replacing \mathcal{D} with \mathcal{D}' and “ $<$ ” with “ \prec ” where appropriate.

We next give a symmetric statement for $S - \{\min S\}$ and $\mathcal{D}_{>\min S}$ as the below corollary:

Corollary A.1. *Given a dataset \mathcal{D} of N unique real values in $[a, b]$ and a uniform without-replacement sample S of m values from \mathcal{D} such that $\min S = a'$, the set $S - \{a'\}$ is a uniform without-replacement sample from $\mathcal{D}_{>a'} = \mathcal{D} \cap (a', b]$, for any applicable value of $a' \in \mathcal{D}$.*

Monotonicity Property and Correctness Proof. Before proving the main result, we briefly describe the *dataset size monotonicity property* obeyed by all bounders in [Chapter 6](#). This fact will be used in the main correctness proof. When N is unknown, an upper bound on N suffices, because bounders in [Chapter 6](#) all satisfy the following: for any S, a, b, N, δ , and $N' > N$,

$$\text{Lbound}(S, a, b, N', \delta) \leq \text{Lbound}(S, a, b, N, \delta) \quad (\text{A.42})$$

$$\text{Rbound}(S, a, b, N', \delta) \geq \text{Rbound}(S, a, b, N, \delta) \quad (\text{A.43})$$

That is, using an upper bound for N can only make the CI looser, and since SSI range-based error bounders with the correct dataset size N fail with probability at most δ , they must also fail with probability at most δ for any $N' > N$.

We are now ready to prove [Theorem 6.2](#) on the correctness of [Algorithm 6.4](#), which we restate below for convenience.

Theorem 6.2. *Given SSI range-based bounders Lbound and Rbound for computing lower (resp. upper) confidence bounds and a dataset \mathcal{D} of N unique values known to all fall in the interval $[a, b] \subseteq \mathbb{R}$, [Algorithm 6.4](#) returns a $(1 - \delta)$ confidence interval for $\text{AVG}(\mathcal{D})$.*

Proof. [Algorithm 6.4](#) proceeds by drawing S uniformly and without replacement from \mathcal{D} and computing $a' = \min S$, $b' = \max S$, S_ℓ , and S_r , where the latter two quantities capture relevant statistics from the sample $S - \{b'\}$ and $S - \{a'\}$, respectively, so we treat S_ℓ and S_r as if $S_\ell = S - \{b'\}$ and $S_r = S - \{a'\}$. By [Lemma A.3](#), we have that S_ℓ is a uniform sample of $m - 1$ values drawn without replacement from $\mathcal{D}_{<b'}$, and likewise by [Corollary A.1](#) S_r is a uniform sample of $m - 1$ values drawn without replacement from $\mathcal{D}_{>a'}$. Because Lbound and Rbound are assumed to be SSI, range-based error bounders, we have that

$$\mathbb{P}\left(\text{Lbound}(S_\ell, a, b', N - 1, \frac{\delta}{2}) > \text{AVG}(\mathcal{D})\right) \quad (\text{A.44})$$

$$\leq \mathbb{P} \left(\text{Lbound}(S_\ell, a, b', |\mathcal{D}_{<b'}|, \frac{\delta}{2}) > \text{AVG}(\mathcal{D}) \right) \quad (\text{A.45})$$

$$\leq \mathbb{P} \left(\text{Lbound}(S_\ell, a, b', |\mathcal{D}_{<b'}|, \frac{\delta}{2}) > \text{AVG}(\mathcal{D}_{<b'}) \right) < \frac{\delta}{2} \quad (\text{A.46})$$

and symmetrically for $\text{Rbound}(S_r, a', b, N-1, \frac{\delta}{2})$, but with “>” replaced with “<” in the probability expression above, and replacing $\mathcal{D}_{<b'}$ with $\mathcal{D}_{>a'}$. (A.44) \rightarrow (A.45) follows from the dataset size monotonicity property of Lbound (§6.2.2), applicable since $N-1 \geq |\mathcal{D}_{<b'}|$, and (A.45) \rightarrow (A.46) follows since $\text{AVG}(\mathcal{D}_{<b'}) \leq \text{AVG}(\mathcal{D})$, as the former is clipped above b' (and similarly for Rbound since $\text{AVG}(\mathcal{D}_{>a'}) \geq \text{AVG}(\mathcal{D})$). Union bounding over the cases for each of Lbound and Rbound , the probability that Algorithm 6.4 returns an interval that does not enclose $\text{AVG}(\mathcal{D})$ is at most δ .

APPENDIX B: HML SPECIFICATIONS

B.1 SUMMARY OF HML SEMANTICS

Phrase	Usage	Operation	Example
refers_to	<i>string</i> refers_to <i>NBSAFETY</i> object	Register a <i>NBSAFETY</i> object 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"
	$DC[E]$ /Scalar results_from <i>clause</i>	Specify the name for <i>clause</i> 's output $DC[E]$.	"learned" results_from "L" on "income"
uses	<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 B.1: 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*.

B.2 HML GRAMMAR

```

<var>      ::= <string>
<scanner>  ::= <var> | <scanner-obj>
<extractor> ::= <var> | <extractor-obj>
<typed-ext> ::= 'C' <var> ' ,' <extractor> ')'
<extractors> ::= 'C' <extractor> { ' ,' <extractor> } ')'
<typed-exts> ::= 'C' <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> | 'C' <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 B.1: HML syntax in Extended Backus-Naur Form. *<string>* denotes a legal String object in Scala; *<*-obj>* denotes the correct syntax for instantiating object of type “*”; *<Scala expr>* 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.

APPENDIX C: SYSTEM EXTENSIONS

C.1 GENERALIZING FASTMATCH TO ADDITIONAL QUERIES

SUM aggregations

While we did not consider it explicitly in [Chapter 5](#), Ding et al. [20] describe how to perform SUM aggregations with ℓ_2 distributional guarantees via *measure-biased sampling*. Briefly, a measure-biased sample for some attribute Y involves sampling each tuple t in T , where the probability of inclusion in the sample is proportional t 's value of Y . FASTMATCH can also leverage measure-biased samples in order to match bar graphs generated via the following types of queries:

```
SELECT X, SUM(Y) FROM T
WHERE Z = z_i GROUP BY X
```

Figure C.1: Template for histogram-generating query involving SUM aggregation.

As in [Definition 5.1](#), Z is the candidate attribute and X is the grouping attribute for the x-axis. One measure-biased sample must be created per measure attribute Y the analyst is interested in, so if there are n such attributes, we require an additional n complete passes over the data for preprocessing. When matching bar graphs generated according to the above template, FASTMATCH would simply use the measure-biased sample for Y and pretend as if it were matching visualizations generated according to [Definition 5.1](#); that is, it would use COUNT instead of SUM. There is nothing special about the ℓ_2 metric used in [20], and the same techniques may be used by FASTMATCH to process queries satisfying [Guarantees 5.1](#) and [5.2](#).

Candidates based off arbitrary boolean predicates

In order to support candidates based off boolean predicates such as $Z^{(1)} = z_i^{(1)} \wedge Z^{(2)} = z_j^{(2)}$, FASTMATCH needs a way to estimate the number of active tuples in a block for the purposes of applying AnyActive block selection. In this case, simple bitmap indexes with one bit per block are not enough. We may instead opt to use the slightly costlier *density maps* from [11]. We refer readers to that paper for a description of how to estimate the number of tuples in a block satisfying an arbitrary boolean predicate. Even if different candidates share some of the same tuples, our guarantees still hold since HistSim uses a Holm-Bonferroni procedure to get joint guarantees across different candidates at a given iteration, a method which is agnostic to any dependency structure between candidates.

Multiple attributes in GROUP BY clause

In the case where the analyst wishes to use multiple attributes $X^{(1)}, X^{(2)}, \dots, X^{(n)}$ to generate the support of our histograms generated via [Definition 5.1](#), all of the same methods apply, but we estimate the support $|V_X|$ as

$$|V_{X^{(1)}}| \cdot |V_{X^{(2)}}| \cdot \dots \cdot |V_{X^{(n)}}| \quad (\text{C.1})$$

This may be an overestimate if two attribute values, say $x_i^{(1)}$ and $x_j^{(2)}$, never occur together. Our guarantees still hold in this case — overestimating the size of the support can only make the bound in [Theorem 5.1](#) looser, which does not cause any correctness issues.

Handling continuous X attributes via binning

If the analyst wishes to use a continuous X , they must simply provide a set of non-overlapping bin ranges, or “buckets” in which to collect tuples. Everything else is still the same. In fact, FLIGHTS-q1 and FLIGHTS-q2 used this technique, since the DepartureHour attribute was actually a continuous attribute we placed into 24 bins (although we presented it as a discrete attribute for simplicity).

Handling an unknown candidate domain

If the candidate domain is unknown at query time, for example if we do not have any bitmap index structures over the attribute(s) used to generate candidates, it is still possible to use a variant of our methods. First of all, we may still employ ScanMatch, creating state for new candidates as they are discovered. During stage 1 of HistSim, in which rare candidates are pruned, we must also account for any potential candidates for which HistSim has not yet seen any tuples. In this case, we may simply add one additional “dummy” candidate which matches against all the tuples for any unseen candidates. We add an additional test to the Holm-Bonferroni procedure for this dummy candidate — if the test rejects, and if U represents the indices of the unseen candidates, then we can be sure that $\frac{\sum_{j \in U} N_j}{N} < \sigma$, which in turn implies that $\frac{N_j}{N} < \sigma$ for each $j \in U$.

Handling continuous candidates

If one or more of the attributes used to group candidates is continuous, then, as in the case of continuous X , candidates may be “grouped” by placing different real-values into bins. We can also construct bitmaps for continuous attributes at some predetermined finest level of granularity of binning, which can then be used to induce bitmaps for any coarser granularity that may be needed. Even if the finest granularity available is too coarse to isolate different candidates, as long as it

isolates some subsets of candidates, it may still be useful for pruning the blocks that need to be considered for AnyActive block selection. Even if there is no index available, one may still use ScanMatch.

C.2 DIFFERENT TYPES OF FASTMATCH GUARANTEES

Allowing Distinct ε_1 and ε_2 for **Guarantees 5.1** and **5.2**

If the analyst believes one of **Guarantees 5.1** and **5.2** is more important than the other, they may indicate this by providing separate ε_1 for **Guarantee 5.1** and ε_2 for **Guarantee 5.2**; HistSim generalizes in a very straightforward way in this case. For example, if **Guarantee 5.2** is more important than **Guarantee 5.1**, the analyst may provide ε_1 and ε_2 with $\varepsilon_2 < \varepsilon_1$.

Allowing other distance metrics

We can extend HistSim to work for any distance metric for which there exists an analogue to **Theorem 5.1**. For example, there exist such bounds for ℓ_2 distance [20, 151].

Allowing a range of k in input

In some cases, the analyst may not care about the exact number of matching candidates. For example, the analyst may be fine with finding anywhere between 5 and 10 of the closest histograms to a target. In this case, they may specify a range $[k_1, k_2]$, and FASTMATCH may return some number $k \in [k_1, k_2]$ of histograms matching the target, where k is automatically picked to make it as easy as possible to satisfy **Guarantees 5.1** and **5.2**. For example, in the case $[k_1, k_2] = [5, 10]$, there may be a very large separation between the 7th- and 8th-closest candidates, in which case HistSim can automatically choose $k = 7$, as this likely provides a small δ^{upper} as soon as possible.

C.3 HANDLING ARBITRARY EXPRESSIONS IN FASTFRAME

Throughout **Chapter 6**, we assumed that column c_i was known to lie in some range $[a_i, b_i]$. We then showed how to feed these bounds into our RangeTrim procedure to compute conservative CIs for, e.g., $\text{AVG}(c_i)$. In general, however, we may want to compute an aggregate involving an arbitrary expression in terms of several columns. That is, we may want to compute CIs for, e.g., $\text{AVG}(f(c_1, \dots, c_n))$. We now show how to do so for a large class of f by optimizing over such

f (while using the individual bounds $[a_i, b_i]$ for each column as constraints) in order to compute *derived* range bounds of the form

$$\left[\inf_{c_1, \dots, c_n} f(c_1, \dots, c_n), \sup_{c_1, \dots, c_n} f(c_1, \dots, c_n) \right] \quad (\text{C.2})$$

Applicable Expressions. To compute a derived lower range bound, we need to be able to either solve or compute a lower bound for the following optimization problem:

$$\min_{c_1, \dots, c_n} f(c_1, \dots, c_n) \quad (\text{C.3})$$

$$\text{s.t. } a_i \leq c_i \leq b_i, \quad \forall 1 \leq i \leq n \quad (\text{C.4})$$

The case for the derived upper range bound is analogous, but with f replaced by $-f$. We show how to compute both lower and upper derived bounds under two kinds of conditions: (i) *the monotonicity condition*; i.e., f is monotone in each c_i , and (ii) *the convexity condition*; i.e., either f or $-f$ is convex. This handles a large number of expressions in practice.

1. *Expressions Monotone in each Column.* If f is monotone in each column c_i , one simply needs to check whether a_i or b_i gives the smaller (resp. larger) value when computing the lower (resp. upper) bound, and evaluate f on the boundaries for each of these cases.

2. *Convex or Concave Expressions.* Without loss of generality, we now consider the case of convex f . A large body of existing work focuses on minimizing a convex function subject to convex constraints; please see Boyd et al. [209] for relevant background. In our case, the constraints are all linear (and are sometimes referred to as “box” constraints), and most kinds of convex functions in practice can be optimized efficiently with off-the-shelf software under such constraints, so we do not go into detail here.

Maximizing a f under box constraints is more difficult. Since f is convex, the maximum (and therefore the derived upper range bound we seek) will occur at some set of boundary points; i.e., if $a_i \leq c_i \leq b_i$, we know that the maximum will occur at one of $c_i = a_i$ or $c_i = b_i$. If we have n columns involved in the expression f , however, we will require evaluating f on all 2^n combinations of boundary points for the constraints. Fortunately, database aggregates over expressions typically do not involve more than 2 or 3 columns, and any $n \leq 20$ or so can be handled without trouble.

Example C.1. Suppose the user issues a query to compute $\text{AVG}((2c_1 + 3c_2 - 1)^2)$ involving columns c_1 and c_2 , where we have range bounds $c_1 \in [-3, 1]$ and $c_2 \in [-1, 3]$. The minimum of $(2c_1 + 3c_2 - 1)^2$ subject to these constraints is simply 0, and can be found via quadratic programming. The maximum can be obtained by checking the boundaries $(-3, -1)$, $(-3, 3)$, $(1, -1)$, and $(1, 3)$,

and we see that it occurs at $(1, 3)$, for which $(2 \cdot 1 + 3 \cdot 3 - 1)^2 = 100$; thus, the derived range bounds will be $[0, 100]$.

REFERENCES

- [1] Z. Liu and J. Heer, “The effects of interactive latency on exploratory visual analysis,” *IEEE TVCG*, vol. 20, no. 12, pp. 2122–2131, 2014.
- [2] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “Online aggregation,” *ACM SIGMOD Record*, vol. 26, no. 2, pp. 171–182, jun 1997. [Online]. Available: <http://dl.acm.org/citation.cfm?id=253262.253291>
- [3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, “Blinkdb: Queries with bounded errors and bounded response times on very large data,” in *EuroSys*. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2465351.2465355> pp. 29–42.
- [4] B. Mozafari and N. Niu, “A handbook for building an approximate query engine,” *IEEE Data Eng. Bull.*, vol. 38, no. 3, pp. 3–29, 2015.
- [5] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfeld, “I’ve seen “enough”: Incrementally improving visualizations to support rapid decision making,” in *VLDB*, 2017.
- [6] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld, “Rapid sampling for visualizations with ordering guarantees,” *PVLDB*, vol. 8, no. 5, pp. 521–532, Jan. 2015. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2735479.2735485>
- [7] N. Potti and J. M. Patel, “Daq: a new paradigm for approximate query processing,” *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 898–909, 2015.
- [8] 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.
- [9] C. Zhang, “Deepdive: A data management system for automatic knowledge base construction,” Ph.D. dissertation, Citeseer, 2015.
- [10] 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>
- [11] A. Kim, L. Xu, T. Siddiqui, S. Huang, S. Madden, and A. Parameswaran, “Optimally leveraging density and locality for exploratory browsing and sampling,” in *Proceedings of the 3rd Workshop on Human-In-the-Loop Data Analytics*, 2018, pp. 1–7.
- [12] D. Koop and J. Patel, “Dataflow notebooks: encoding and tracking dependencies of cells,” in *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.
- [13] K. Zielnicki, *Nodebook*, 2017 (accessed July 5, 2020), <https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/>.

- [14] *Datalore*, 2018 (accessed December 1, 2020), <https://datalore.jetbrains.com/>.
- [15] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, “Effortless data exploration with zenvisage: an expressive and interactive visual analytics system,” *PVLDB*, vol. 10, no. 4, pp. 457–468, 2016.
- [16] J. T. Behrens, “Principles and procedures of exploratory data analysis,” *Psychological Methods*, vol. 2, no. 2, p. 131, 1997.
- [17] P. Hanrahan, “Analytic database technologies for a new kind of user: The data enthusiast,” in *SIGMOD*. New York, NY, USA: ACM, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2213836.2213902> pp. 577–578.
- [18] P. Pirolli and S. Card, “The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis,” in *Proceedings of international conference on intelligence analysis*, vol. 5, 2005, pp. 2–4.
- [19] A. Agarwal, S. Agarwal, S. Assadi, and S. Khanna, “Learning with limited rounds of adaptivity: Coin tossing, multi-armed bandits, and ranking from pairwise comparisons,” in *Conference on Learning Theory*, 2017, pp. 39–75.
- [20] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang, “Sample + seek: Approximating aggregates with distribution precision guarantee,” in *SIGMOD*, 2016.
- [21] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, “Overcoming limitations of sampling for aggregation queries,” in *ICDE*. IEEE, 2001, pp. 534–542.
- [22] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, “Knowing when you’re wrong,” in *SIGMOD*. New York, New York, USA: ACM Press, June 2014. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2588555.2593667> pp. 481–492.
- [23] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo, “The analytical bootstrap: a new method for fast error estimation in approximate query processing,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 277–288.
- [24] D. Alabi and E. Wu, “Pfunk-h: approximate query processing using perceptual models,” in *Proceedings of the 1st Workshop on Human-In-the-Loop Data Analytics*, 2016, p. 10.
- [25] T. Kluyver et al., “Jupyter notebooks-a publishing format for reproducible computational workflows,” in *ELPUB*, 2016, pp. 87–90.
- [26] J. Grus, *I Don’t Like Notebooks (JupyterCon 2018 Talk)*, 2018 (accessed June 26, 2020), <https://t.ly/Wt3S>.
- [27] D. Xin, S. Macke, L. Ma, R. Ma, S. Song, and A. Parameswaran, “Helix: Holistic optimization for accelerating iterative machine learning,” *arXiv preprint arXiv:1812.05762*, 2018.

- [28] S. Macke, A. Beutel, T. Kraska, M. Sathiamoorthy, D. Z. Cheng, and E. Chi, “Lifting the curse of multidimensional data with learned existence indexes,” in *Workshop on ML for Systems at NeurIPS*, 2018.
- [29] S. Macke, Y. Zhang, S. Huang, and A. Parameswaran, “Adaptive sampling for rapidly matching histograms,” *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1262–1275, 2018.
- [30] S. Macke, M. Aliakbarpour, I. Diakonikolas, A. Parameswaran, and R. Rubinfeld, “Rapid approximate aggregation with distribution-sensitive interval guarantees,” in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1703–1714.
- [31] S. Macke, H. Gong, D. J.-L. Lee, A. Head, D. Xin, and A. Parameswaran, “Fine-grained lineage for safer notebook interactions,” *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1093–1101, 2021.
- [32] 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.
- [33] 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.
- [34] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhvani, 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] S. Owen, R. Anil, T. Dunning, and E. Friedman, “Mahout in action,” 2012.
- [39] 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.
- [40] 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,” *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

- [41] J. Langford, L. Li, and A. Strehl, “Vowpal wabbit online learning project,” 2007.
- [42] 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.
- [43] 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.
- [44] A. Damien et al., “Tflearn,” 2016.
- [45] 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.
- [46] D. Team, “Deeplearning4j: Open-source distributed deep learning for the jvm,” *Apache Software Foundation License*, vol. 2, 2016.
- [47] 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.
- [48] 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/>.
- [49] J. Barnes, “Azure machine learning microsoft azure essentials,” 2015.
- [50] “Meet michelangelo: Uber’s machine learning platform,” 2017 (accessed October 8, 2018), <https://eng.uber.com/michelangelo/>.
- [51] *Amazon SageMaker Developer Guide*, 2018 (accessed October 8, 2018), <https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf>.
- [52] 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>.
- [53] M. Beauchemin, “Airflow: a workflow management platform,” 2015 (accessed October 8, 2018), <https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8>.
- [54] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias, “Selection and pruning algorithms for bitmap index selection problem using data mining,” in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2007, pp. 221–230.
- [55] R. Wrembel, *Data Warehouses and OLAP: Concepts, Architectures and Solutions: Concepts, Architectures and Solutions*. Igi Global, 2006.

- [56] D. Lemire, O. Kaser, and K. Aouiche, “Sorting improves word-aligned bitmap indexes,” *Data & Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.
- [57] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, “Consistently faster and smaller compressed bitmaps with roaring,” *Software: Practice and Experience*, vol. 46, no. 11, pp. 1547–1569, 2016.
- [58] S. Chambi, D. Lemire, O. Kaser, and R. Godin, “Better bitmap performance with roaring bitmaps,” *Software: practice and experience*, vol. 46, no. 5, pp. 709–719, 2016.
- [59] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.
- [60] A. Colantonio and R. Di Pietro, “Concise: Compressed ‘n’composable integer set,” *Information Processing Letters*, vol. 110, no. 16, pp. 644–650, 2010.
- [61] K. Wu, K. Stockinger, and A. Shoshani, “Breaking the curse of cardinality on bitmap indexes,” in *Scientific and Statistical Database Management*. Springer, 2008, pp. 348–365.
- [62] F. Fusco, M. P. Stoecklin, and M. Vlachos, “Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1382–1393, 2010.
- [63] M. Mitzenmacher, “A model for learned bloom filters and related structures,” *arXiv preprint arXiv:1802.00884*, 2018.
- [64] M. Mitzenmacher, “Optimizing learned bloom filters by sandwiching,” *arXiv preprint arXiv:1803.01474*, 2018.
- [65] B. Efron et al., “Bootstrap methods: Another look at the jackknife,” *The Annals of Statistics*, vol. 7, no. 1, pp. 1–26, 1979.
- [66] B. Efron, “Bootstrap methods: another look at the jackknife,” in *Breakthroughs in statistics*. Springer, 1992, pp. 569–593.
- [67] Student, “The probable error of a mean,” *Biometrika*, pp. 1–25, 1908.
- [68] J. Hájek, “Limiting distributions in simple random sampling from a finite population,” *Publications of the Mathematics Institute of the Hungarian Academy of Science*, vol. 5, pp. 361–74, 1960.
- [69] A. Pol and C. Jermaine, “Relational confidence bounds are easy with the bootstrap,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 587–598.
- [70] B. Mozafari, “Approximate query engines: Commercial challenges and research opportunities,” in *SIGMOD*. ACM, 2017, pp. 521–524.
- [71] K. Li and G. Li, “Approximate query processing: What is new and where to go?” *Data Science and Engineering*, vol. 3, no. 4, pp. 379–397, 2018.

- [72] D. Fisher, “Incremental, approximate database queries and uncertainty for exploratory visualization,” in *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 2011, pp. 73–80.
- [73] B. C. Kwon, J. Verma, P. J. Haas, and C. Demiralp, “Sampling for scalable visual analytics,” *IEEE computer graphics and applications*, vol. 37, no. 1, pp. 100–108, 2017.
- [74] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, “The aqua approximate query answering system,” in *ACM Sigmod Record*, vol. 28, no. 2. ACM, 1999, pp. 574–576.
- [75] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica, “Blink and it’s done: interactive queries on very large data,” 2012.
- [76] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra, “Scalable approximate query processing with the dbo engine,” *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 4, p. 23, 2008.
- [77] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, “Verdictdb: Universalizing approximate query processing,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1461–1476.
- [78] V. Ganti, M.-L. Lee, and R. Ramakrishnan, “Icicles: Self-tuning samples for approximate query answering,” in *VLDB*, vol. 176, no. 187, 2000.
- [79] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska, “Vistrees: fast indexes for interactive data exploration,” in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM, 2016, p. 5.
- [80] L. Valiant, *Probably Approximately Correct: Nature’s Algorithms for Learning and Prospering in a Complex World*. Basic Books (AZ), 2013.
- [81] D. Moritz, D. Fisher, B. Ding, and C. Wang, “Trust, but verify: Optimistic visualizations of approximate queries for exploring big data,” in *CHI*. ACM, 2017, pp. 2904–2915.
- [82] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri, “Efficient sampling strategies for relational database operations,” *Theoretical Computer Science*, vol. 116, no. 1, pp. 195–226, 1993.
- [83] S. Chattopadhyay et al., “What’s wrong with computational notebooks? pain points, needs, and design opportunities,” in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–12.
- [84] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, “Managing messes in computational notebooks,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.
- [85] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, “The story in the notebook: Exploratory data science using a literate programming tool,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–11.

- [86] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo, “The design space of computational notebooks: An analysis of 60 systems in academia and industry,” in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, ser. VL/HCC ’20, Aug 2020.
- [87] J. M. Perkel, “Why jupyter is data scientists’ computational notebook of choice,” *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.
- [88] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, “A large-scale study about quality and reproducibility of jupyter notebooks,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 507–517.
- [89] A. Rule, A. Tabard, and J. D. Hollan, “Exploration and explanation in computational notebooks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.
- [90] M. K. Anand, S. Bowers, T. Mcphillips, and B. Ludäscher, “Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs,” in *Scientific and Statistical Database Management*. Springer, 2009, pp. 237–254.
- [91] S. Bowers, “Scientific workflow, provenance, and data modeling challenges and approaches,” *Journal on Data Semantics*, vol. 1, no. 1, pp. 19–30, 2012.
- [92] S. B. Davidson, S. C. Boulakia et al., “Provenance in scientific workflow systems,” *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 44–50, 2007.
- [93] S. B. Davidson and J. Freire, “Provenance and scientific workflows: challenges and opportunities,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1345–1350.
- [94] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan, “Archiving scientific data,” *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 1, pp. 2–42, 2004.
- [95] T. J. Green, G. Karvounarakis, and V. Tannen, “Provenance semirings,” in *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’07. New York, NY, USA: ACM, 2007. [Online]. Available: <http://doi.acm.org/10.1145/1265530.1265535> pp. 31–40.
- [96] J. Cheney, L. Chiticariu, and W.-C. Tan, *Provenance in databases: Why, how, and where*. Now Publishers Inc, 2009.
- [97] M. Herschel et al., “A survey on provenance: What for? what form? what from?” *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.
- [98] P. J. Guo and M. I. Seltzer, “Burrito: Wrapping your lab notebook in computational infrastructure,” 2012.

- [99] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, “noworkflow: capturing and analyzing provenance of scripts,” in *International Provenance and Annotation Workshop*. Springer, 2014, pp. 71–83.
- [100] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, “noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, 2017.
- [101] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, “Collecting and analyzing provenance on interactive notebooks: when ipython meets noworkflow,” in *7th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 15)*, 2015.
- [102] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, and J. Freire, “Your notebook is not crumby enough, replace it.” in *CIDR*, 2020.
- [103] 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.
- [104] 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.
- [105] D. Team et al., “Deeplearning4j: Open-source distributed deep learning for the jvm,” *Apache Software Foundation License*, vol. 2.
- [106] 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.
- [107] 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.
- [108] 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. [Online]. Available: <http://doi.acm.org/10.1145/1142473.1142552> pp. 706–706.
- [109] J. Rosen, “Pyspark internals.”
- [110] 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.
- [111] “Scikit-learn user guide,” http://scikit-learn.org/stable/user_guide.html.

- [112] 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.
- [113] A. Paszke, S. Gross, S. Chintala, and G. Chanan, “Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration,” 2017.
- [114] 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.
- [115] 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.
- [116] 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.
- [117] 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.
- [118] A. M. Pitts, “Operationally-based theories of program equivalence,” *Semantics and Logics of Computation*, vol. 14, p. 241, 1997.
- [119] A. D. Gordon, “A tutorial on co-induction and functional programming,” in *Functional Programming, Glasgow 1994*. Springer, 1995, pp. 78–95.
- [120] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education, 2006.
- [121] 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.
- [122] “Deepdive census example,” <https://github.com/HazyResearch/deepdive/tree/master/examples/census>.
- [123] 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.
- [124] D. Dheeru and E. Karra Taniskidou, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [125] 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.
- [126] 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.

- [127] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
- [128] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 489–504.
- [129] “Supercharging the Git Commit Graph IV: Bloom Filters,” <https://blogs.msdn.microsoft.com/devops/2018/07/16/super-charging-the-git-commit-graph-iv-bloom-filters/>, 2018, accessed: July 2018.
- [130] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [131] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [132] Y. She and A. B. Owen, “Outlier detection using nonconvex penalized regression,” *Journal of the American Statistical Association*, vol. 106, no. 494, pp. 626–639, 2011.
- [133] J. Tibshirani and C. D. Manning, “Robust logistic regression using shift parameters (long version),” *arXiv preprint arXiv:1305.4987*, 2013.
- [134] “Flight Records,” <http://stat-computing.org/dataexpo/2009/the-data.html>, 2009.
- [135] H. Wickham, *ggplot2: elegant graphics for data analysis*. Springer, 2016.
- [136] M. Bostock, V. Ogievetsky, and J. Heer, “D³ data-driven documents,” *IEEE TVCG*, vol. 17, no. 12, pp. 2301–2309, 2011.
- [137] C. Stolte, D. Tang, and P. Hanrahan, “Polaris: A system for query, analysis, and visualization of multidimensional relational databases,” *IEEE TVCG*, vol. 8, no. 1, pp. 52–65, 2002.
- [138] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis, “Seedb: efficient data-driven visualization recommendations to support visual analytics,” *PVLDB*, vol. 8, no. 13, pp. 2182–2193, 2015.
- [139] T. Batu, L. Fortnow, R. Rubinfeld, W. D. Smith, and P. White, “Testing that distributions are close,” in *FOCS*, 2000.
- [140] A. L. Gibbs and F. E. Su, “On choosing and bounding probability metrics,” *International statistical review*, vol. 70, no. 3, pp. 419–435, 2002.
- [141] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 1979, pp. 23–34.

- [142] M. S. Kester, M. Athanassoulis, and S. Idreos, “Access path selection in main-memory optimized data systems: Should i scan or should i probe?” in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 715–730.
- [143] G. Casella and R. L. Berger, *Statistical inference*. Duxbury Pacific Grove, CA, 2002, vol. 2.
- [144] E. L. Lehmann and J. P. Romano, *Testing statistical hypotheses*. Springer Science & Business Media, 2006.
- [145] S. Holm, “A simple sequentially rejective multiple test procedure,” *Scandinavian journal of statistics*, pp. 65–70, 1979.
- [146] N. L. Johnson, A. W. Kemp, and S. Kotz, *Univariate discrete distributions*. John Wiley & Sons, 2005, vol. 444.
- [147] I. Diakonikolas, Personal communication, 2017.
- [148] C. McDiarmid, “On the method of bounded differences,” *Surveys in combinatorics*, vol. 141, no. 1, pp. 148–188, 1989.
- [149] S.-O. Chan, I. Diakonikolas, G. Valiant, and P. Valiant, “Optimal algorithms for testing closeness of discrete distributions,” in *SODA*, 2014, pp. 1193–1203.
- [150] C. Daskalakis, I. Diakonikolas, R. O’Donnell, R. A. Servedio, and L.-Y. Tan, “Learning sums of independent integer random variables,” in *FOCS*. IEEE, 2013, pp. 217–226.
- [151] B. Waggoner, “L p testing and learning of discrete distributions,” in *ITCS*. ACM, 2015, pp. 347–356.
- [152] “Boost Statistical Distributions and Functions,” https://www.boost.org/doc/libs/1_67_0/libs/math/doc/html/dist.html, 2006.
- [153] C.-Y. Chan and Y. E. Ioannidis, “Bitmap index design and evaluation,” in *ACM SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 355–366.
- [154] K. Wu, E. Otoo, and A. Shoshani, “Compressed bitmap indices for efficient query processing,” *Lawrence Berkeley National Laboratory*, 2001.
- [155] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” *Journal of the American statistical association*, vol. 58, no. 301, pp. 13–30, 1963.
- [156] R. Bardenet, O.-A. Maillard et al., “Concentration inequalities for sampling without replacement,” *Bernoulli*, vol. 21, no. 3, pp. 1361–1385, 2015.
- [157] S. Wu, B. C. Ooi, and K.-L. Tan, “Continuous sampling for online aggregation over multiple queries,” in *SIGMOD*. ACM, 2010, pp. 651–662.
- [158] C. Qin and F. Rusu, “Pf-ola: a high-performance framework for parallel online aggregation,” *Distributed and Parallel Databases*, vol. 32, no. 3, pp. 337–375, 2014.

- [159] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica, “G-ola: Generalized on-line aggregation for interactive analysis on big data,” in *SIGMOD*. ACM, 2015, pp. 913–918.
- [160] B. Nichols, D. Buttler, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.
- [161] “NYC Taxi Trip Records,” <https://github.com/toddwscneider/nyc-taxi-data/>, 2015.
- [162] “WA Police Stop Records,” <https://stacks.stanford.edu/file/druid:py883nd2578/WA-clean.csv.gz>, 2017.
- [163] R. J. Serfling, “Probability inequalities for the sum in sampling without replacement,” *The Annals of Statistics*, pp. 39–48, 1974.
- [164] M. Wainwright, “Basic tail and concentration bounds,” URL: https://www.stat.berkeley.edu/.../Chap2_TailBounds_Jan22_2015.pdf (visited 12/31/2017), 2015.
- [165] F. Olken, “Random sampling from databases,” Ph.D. dissertation, University of California, Berkeley, 1993.
- [166] “Microsoft sql server 2019 documentation: Clr user-defined aggregates – requirements,” <https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-user-defined-aggregates-requirements?view=sql-server-ver15>, 2017, Date accessed: 2020-02-27.
- [167] “Oracle documentation: Using user-defined aggregate functions,” https://docs.oracle.com/cd/B28359_01/appdev.111/b28425/aggr_functions.htm, 2020, Date accessed: 2020-02-24.
- [168] “Postgresql documentation: User-defined aggregates,” <https://www.postgresql.org/docs/12/xaggr.html>, 2020, Date accessed: 2020-02-24.
- [169] S. Gupta, S. Purandare, and K. Ramachandra, “Aggify: Lifting the curse of cursor loops using custom aggregates,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 559–573.
- [170] B. Welford, “Note on a method for calculating corrected sums of squares and products,” *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
- [171] T. F. Chan, G. H. Golub, and R. J. LeVeque, “Algorithms for computing the sample variance: Analysis and recommendations,” *The American Statistician*, vol. 37, no. 3, pp. 242–247, 1983.
- [172] R. F. Ling, “Comparison of several algorithms for computing sample means and variances,” *Journal of the American Statistical Association*, vol. 69, no. 348, pp. 859–866, 1974.
- [173] A. Dvoretzky, J. Kiefer, J. Wolfowitz et al., “Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator,” *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 642–669, 1956.

- [174] T. W. Anderson, “Confidence limits for the expected value of an arbitrary bounded random variable with a continuous distribution function,” STANFORD UNIV CA DEPT OF STATISTICS, Tech. Rep., 1969.
- [175] P. Massart, “The tight constant in the dvoretzky-kiefer-wolfowitz inequality,” *The annals of Probability*, pp. 1269–1283, 1990.
- [176] P. J. Haas, *Hoeffding inequalities for join-selectivity estimation and online aggregation*. IBM, 1996.
- [177] C. Chen, W. Wang, X. Wang, and S. Yang, “Effective order preserving estimation method,” in *Australasian Database Conference*. Springer, 2016, pp. 369–380.
- [178] 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*, 2012, pp. 325–336.
- [179] D. Lemire, “External-memory shuffling in linear time?” 2010 (accessed August 12, 2020), <https://lemire.me/blog/2010/03/15/external-memory-shuffling-in-linear-time/>.
- [180] P. J. Haas, “Large-sample and deterministic confidence intervals for online aggregation,” in *Proceedings. Ninth International Conference on Scientific and Statistical Database Management (Cat. No. 97TB100150)*. IEEE, 1997, pp. 51–62.
- [181] P. J. Haas and J. M. Hellerstein, “Ripple joins for online aggregation,” *ACM SIGMOD Record*, vol. 28, no. 2, pp. 287–298, 1999.
- [182] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.
- [183] A. Wald, *Sequential analysis*. Courier Corporation, 2004.
- [184] S. Zhao, E. Zhou, A. Sabharwal, and S. Ermon, “Adaptive concentration inequalities for sequential decision problems,” in *Advances in Neural Information Processing Systems*, 2016, pp. 1343–1351.
- [185] R. J. Lipton, J. F. Naughton, and D. A. Schneider, *Practical selectivity estimation through adaptive sampling*. ACM, 1990, vol. 19, no. 2.
- [186] R. J. Lipton and J. F. Naughton, “Estimating the size of generalized transitive closures,” in *Proceedings of the 15th Int. Conf. on Very Large Data Bases*, 1989.
- [187] D. E. Knuth, “Literate programming,” *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.
- [188] H. Shen, “Interactive notebooks: Sharing the code,” *Nature*, vol. 515, no. 7525, pp. 151–152, 2014.

- [189] C. Yan and Y. He, “Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1539–1554.
- [190] J. Ormond, *ACM Recognizes Innovators Who Have Shaped the Digital Revolution*, 2018 (accessed June 26, 2020), <https://awards.acm.org/binaries/content/assets/press-releases/2018/may/technical-awards-2017.pdf>.
- [191] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” *Addison wesley*, vol. 7, no. 8, p. 9, 1986.
- [192] A. Møller and M. I. Schwartzbach, “Static program analysis,” *Notes. Feb*, 2012.
- [193] “sys: System-specific parameters and functions,” <https://docs.python.org/2/library/sys.html#sys.settrace>, 2020, Date accessed: 2020-07-29.
- [194] S. Steegen, F. Tuerlinckx, A. Gelman, and W. Vanpaemel, “Increasing transparency through a multiverse analysis,” *Perspectives on Psychological Science*, vol. 11, no. 5, pp. 702–712, 2016.
- [195] “Github search api,” <https://developer.github.com/v3/search/>, 2020, Date accessed: 2020-07-29.
- [196] “2to3: Automated python 2 to 3 code translation,” <https://docs.python.org/3/library/2to3.html>, 2020, Date accessed: 2020-07-29.
- [197] S. Macke, *NBSafety Experiments*, 2020 (accessed July 29, 2020), <https://github.com/nbsafety-project/nbsafety-experiments/>.
- [198] M. B. Kery and B. A. Myers, “Interactions for untangling messy history in a computational notebook,” in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 147–155.
- [199] M. B. Kery, A. Horvath, and B. A. Myers, “Variolite: Supporting exploratory programming by data scientists.” in *CHI*, vol. 10, 2017, pp. 3 025 453–3 025 626.
- [200] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert, “Altair: Interactive statistical visualizations for python,” *Journal of open source software*, vol. 3, no. 32, p. 1057, 2018.
- [201] M. Bendre, B. Sun, D. Zhang, X. Zhou, K. C. Chang, and A. Parameswaran, “Dataspread: Unifying databases and spreadsheets,” in *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 8, no. 12. NIH Public Access, 2015, p. 2000.
- [202] M. Bendre, T. Wattanawaroon, K. Mack, K. Chang, and A. Parameswaran, “Anti-freeze for large and complex spreadsheets: Asynchronous formula computation,” in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1277–1294.

- [203] 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.
- [204] N. Shavit and D. Touitou, “Software transactional memory,” *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.
- [205] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- [206] M. Yannakakis, “On a class of totally unimodular matrices,” *Mathematics of Operations Research*, vol. 10, no. 2, pp. 280–304, 1985.
- [207] 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.
- [208] A. Wald, J. Wolfowitz et al., “Confidence limits for continuous distribution functions,” *The Annals of Mathematical Statistics*, vol. 10, no. 2, pp. 105–118, 1939.
- [209] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004. [Online]. Available: <https://books.google.com/books?hl=en&lr=&id=IUZdAAAAQBAJ&pgis=1>