Multiversion Hindsight Logging for Iterative AI/ML

Rolando Garcia
UC Berkeley
rogarcia@berkeley.edu

Anusha Dandamudi
UC Berkeley
adandamudi@berkeley.edu

Gabriel Matute
UC Berkeley
gmatute@berkeley.edu

Lehan Wan
UC Berkeley
wan_lehan@berkeley.edu

Joseph Gonzalez
UC Berkeley
jegonzal@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Koushik Sen
UC Berkeley
ksen@berkeley.edu

ABSTRACT

FlorDB is a novel Multiversion Hindsight Logging system designed to support the iterative, exploratory nature of machine learning engineering (MLE). It facilitates the management and examination of extensive ML experiment histories while maintaining the high-speed experimentation loop crucial to MLEs. At its core, FlorDB employs an SQL-like query interface, representing each Python logging statement as a "virtual column" and visualizing each Python code run as a row. Three principal contributions characterize this system: First, the use of the GumTree algorithm for aligning code blocks in logging statement propagation, enabling FlorDB to detect code version changes and propagate logging statements for consistency. Second, the development of a unified relational model for managing historical queries, providing a comprehensive, accessible view of ML experiments over time, thus facilitating advanced data analysis, trend detection, and hypothesis testing. Last, the definition of "physical operators" for efficient query execution and resource management ensures effective system performance, even when dealing with complex queries. By streamlining these processes, FlorDB boosts MLEs' agility and productivity in fast-paced experimentation scenarios, offering a comprehensive, efficient solution for multiversion logging and management in MLE.

PVLDB Reference Format:
Rolando Garcia, Anusha Dandamudi, Gabriel Matute, Lehan Wan, Joseph Gonzalez, Joseph M. Hellerstein, and Koushik Sen. Multiversion Hindsight Logging for Iterative AI/ML. PVLDB, V(I): pp-pp, 2022.
doi:10.14777/3433333.3436925

1 INTRODUCTION

In recent years, model developers have rallied behind agile software engineering practices to quickly navigate the vast space of model architectures and corresponding hyper-parameters with a reasonable chance of success. They rely less on first principles and more on trial-and-error to improve model performance. To quote a machine learning engineer in a recent interview study, "people will have principled stances or intuitions for why models should work, but the most important thing to do is achieve scary high experimentation velocity" [18]. Sustaining high velocity over the course of model development and tuning leads to an explosion in the number of model versions. As organizations grow and move to operationalize machine learning, they confront increasingly more cumbersome data management problems arising from the many versions of model training at scale [9, 21].

Indeed, the agile practices employed in machine learning engineering demand meticulous experiment version management. This is paramount to enable capabilities such as model rollbacks, which are crucial for addressing performance issues, diagnostics, facilitating troubleshooting, and model refinement, as well as retraining to improve model performance with additional data or modified parameters. High-speed experimentation drives innovation in the machine learning landscape, but it needs to be balanced with careful management of artifacts and versions. This ensures that MLEs can optimize their results, maintain high-quality models, and swiftly react to changes or issues, thereby fostering a productive and efficient development cycle.

1.1 Background: Hindsight Logging

Hindsight logging [8] represents an innovative approach to logging that underscores the agility inherent in the practices of Machine Learning Engineers (MLEs). Hindsight logging is a lightweight record-replay technique that allows MLEs to add logging statements to a Python program, and then incrementally *replay* parts of the program very quickly to generate the log outputs that would have emerged had the statements been in the code originally. This technique encourages MLEs to optimistically and flexibly operate with a minimal initial logging scheme, typically restricted to loss and accuracy metrics during the training phase. Additional logging is only implemented when evidence of issues arises (i.e. in deployment) and further key information is required for debugging. In essence, hindsight logging aligns with the rapid-iteration mindset of MLEs, facilitating faster experiment cycles and enabling the swift movement from ideation to model training without regret, even for long-running training processes.

The concept of hindsight logging has been operationalized through systems such as Flor, a record-replay system developed to bolster this logging strategy for model training [8]. Flor embodies

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. V, No. I
ISSN 2150–8097.
doi:10.14777/3433333.3436925

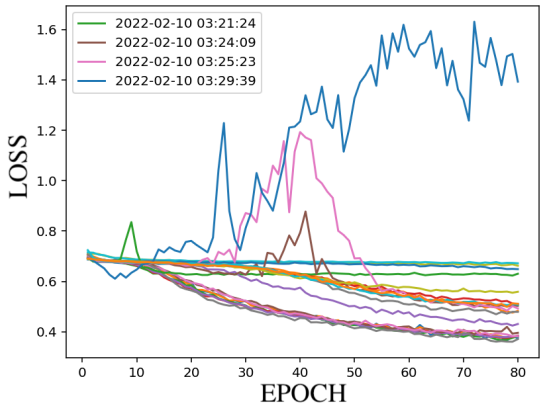


Figure 1: Loss landscape showing the many model versions that are normally explored as a result of model training and fine-tuning. Versions are seemingly similar, splitting into approximately three clusters and a handful of anomalies.

two salient features: background and adaptive checkpointing, and memoization with parallelism during replay. The background and adaptive checkpointing facilitate low-overhead *record*, limiting the computational resources expended during model training. Simultaneously, the system ensures reduced latency during replay by leveraging memoization and parallelism through checkpoint-resume. Thus, Flor provides a robust framework in support of hindsight logging, further enhancing the agility of MLEs in their pursuit of high-velocity experimentation in machine learning.

1.2 FlorDB: Multiversion Hindsight Logging

Hindsight Logging is a useful core technology, but insufficient on its own to address typical ML workflows, which invariably include many training runs. MLEs, in their pursuit of agile and high-velocity experimentation, often wish to revisit not just the most recent run of an experiment, but also prior runs that were performed using different versions of code and data. This presents a complex challenge that goes beyond the capabilities of traditional logging and debugging tools.

To tackle this challenge, in this paper we introduce a novel system, FlorDB, which builds upon and significantly expands the capabilities of Flor. FlorDB is designed to track and manage multiple versions of ML experiments. In doing so, FlorDB provides a more comprehensive and effective solution to hindsight logging, ensuring that MLEs can readily look back through their iterative development processes, thereby better understanding and learning from their experimentation history. Achieving these goals requires overcoming a set of technical challenges described below.

1.2.1 Motivating Scenario. Consider the scenario in which Alice, an MLE, is fine-tuning an object detection model. Over several iterations and numerous versions of her code, she has been logging basic metrics such as loss and accuracy (Figure 1). However, she hasn't been tracking the bounding boxes generated by the model. During a particular iteration, she notices an unexpected change in model performance and suspects that the bounding boxes produced by the model might have drifted significantly.

In an ideal scenario, Alice would like to go back through the history of her experiment, locate the first version where the bounding boxes shifted beyond a certain threshold, and correlate this event with the corresponding changes in model performance. However, she didn't log the bounding boxes from the beginning, creating a gap in her historical data. Given the countless versions of her experiment, it would be impractical and prohibitively time-consuming for Alice to manually revisit past versions, add the necessary logging statements, replay the training for each version, and sift through the results.

Alice's predicament underscores the need for a more efficient and automated approach to hindsight logging, capable of retroactively logging data from past iterations. A more tractable interface would be for Alice to simply add logging statements to her latest code version, and have those changes retroactively applied to all historical versions. Ideally some software could take her change to the latest version, propagate it back across versions, and quickly rerun each version. This would effectively "rewind" time, seamlessly tracking the bounding boxes from each run and version back to the beginning of her experimentation history. Alice could then query the output of those runs in a high-level interface to find the version of interest. This capability would significantly streamline the process of tracking down the root causes of model behavior changes and ultimately improve the agility and effectiveness of ML experimentation.

1.3 Contributions

FlorDB, a Multiversion Hindsight Logging system, provides a SQL-like query interface for ease of use, treating each logged statement as a "virtual column" and every version run as a row. It includes automatic version control and is designed to align with leading ML experiment management tools. In addressing the inherent challenges of implementing Multiversion Hindsight Logging, FlorDB makes the following key contributions:

(1) **Semantic Version Control and Backpropagation via Code Block Alignment**: FlorDB implements an innovative system for the propagation of new logging statements across previous versions. It does so by reassessing methodologies from the field of Software Engineering and leveraging the GumTree algorithm for large-scale code patching. This approach ensures the intelligent propagation of logging statements despite significant code changes over time, tackling the first challenge.
(2) **Historical Query Manager using a Unified Relational Model**: In response to the second challenge of managing historical queries, FlorDB employs a unified relational model. The relational model provides a "virtual table" abstraction, representing a comprehensive view of the log history and facilitating the exploration of the code's complex historical landscape. The unified relational model allows users to frame complex queries using familiar SQL or pandas, thereby simplifying the process of interrogating historical data.

Table 1: Symbol Table

Name	Sym.	Description
ProjectID	p	Name or universal id of the project.
VersionID	$v, \hat{v}$	String hash or timestamp identifying a past run or version of the project. $\hat{v}$ denotes $parent(v)$.
Filename	n	Name of the file.
Text	t	Plain text of the source code file.
Args	a	CLI Arguments passed in to the run.
LineNo	l	Line number in source code from which the record originated.
Iteration	i	Integer index denoting the i^{th} call or instance.
LogLevel	g	Log level denoted by data prep d, outer loop o, or inner loop i.
Epoch	e	Integer index (≥ 1) of the main loop. Not valid for log level data prep d: logging statements that precede the main loop.
Step	s	Integer index (≥ 0) of the nested loop. Only valid for log level inner loop i.
Type	y	Runtime type of the value.
Blob	b	Binary blob of the value.
Name	m	Name of the value.
Val	x	Runtime value.

(3) **Efficient Query Execution and Resource Management via Calibrated Physical Operators**: Addressing the third challenge, FlorDB introduces calibrated physical operators for replay queries. These operators not only optimize query performance but also offer highly accurate wall-clock cost estimates. This feature enhances the user experience by providing meaningful guidance in resource utilization and query formulation.

These contributions position FlorDB as a unique tool in the field, offering advanced solutions for the critical challenges associated with the implementation of Multiversion Hindsight Logging.

2 BACKGROUND & RELATED WORK

Hindsight logging is a natural process in the workflow of Machine Learning Engineers (MLEs), allowing for agile and iterative approaches to training models. By initially logging only the most essential metrics, such as loss and accuracy, MLEs are able to run training jobs swiftly and flexibly. When potential issues arise, additional logging statements can be added and the training jobs rerun to gather more granular data, a practice made more efficient with the use of systems like Flor, a record-replay system supporting low-overhead *record* and low latency *replay* [8]. However, the challenge of applying hindsight logging across multiple versions of training runs, and efficiently managing and querying these logs, is an area that has seen less exploration. The concept of Multiversion Hindsight Logging is novel and builds upon these foundational practices but introduces additional complexities and considerations, which our proposed system, FlorDB, aims to address.

ModelDB [19] is relevant work in the field of machine learning experiment tracking and versioning. Developed by Vartak et al., ModelDB is a system designed to manage machine learning models, providing capabilities to store, version, and manage ML models effectively. The system allows users to log complete model metadata, including hyperparameters, data splits, evaluation metrics, and the final model file. Moreover, it provides the ability to query over this metadata, making it an effective tool for post-hoc analysis and comparison of different ML models and experiment runs.

However, the focus of ModelDB is mainly on managing metadata of deployed models, and it does not emphasize heavily on logging during the model training process, such as the evolving need of MLEs to log and track intermediate results (e.g. bounding boxes) as the developer iterates. Moreover, ModelDB does not inherently support the concept of hindsight logging, i.e., adding additional logging statements post-hoc and backtracking the changes over different versions of code and data. Thus, while ModelDB provides a valuable toolset for model management and versioning, it doesn't fully address the agility and flexibility required in the high-velocity experimentation context of modern MLEs, a gap that our work with FlorDB aims to fill.

MLFlow [22], is another significant project in the domain of machine learning experiment management. MLFlow is an open-source platform that helps manage the end-to-end machine learning lifecycle, including experimentation, reproducibility, and deployment. It provides functionalities to log parameters, code versions, metrics, and output artifacts from each run and later query them. Its modular design allows it to be used with any existing ML library and development process.

However, MLFlow, similar to ModelDB, focuses primarily on tracking completed experiment runs and their corresponding metadata. While it does support logging during training, it does not inherently provide mechanisms for adding logging statements retrospectively, or for propagating such statements to prior code versions – the key tenets of hindsight logging. Also, it does not readily offer the ability to efficiently query and manage large volumes of historical log data across many versions of code and data, a requirement often encountered in high-velocity ML experimentation. Our proposed system, FlorDB, therefore, extends the experiment management paradigm, providing a more dynamic and comprehensive solution to accommodate the evolving needs of agile MLEs.

Acquisitional Query Processing (AQP), proposed by Samuel Madden et al. [14], brings a novel perspective to the world of query processing. Traditional query processing assumes that data is pre-stored and ready for querying, but AQP introduces the concept of acquiring data as a part of the query process. The idea is particularly useful for applications like sensor networks, where querying can be expensive in terms of energy or computational resources.

The fundamental premise of AQP parallels the concept of Multiversion Hindsight Logging, where queries are not just made over pre-stored data, but also consider acquiring (or re-acquiring) data through experiment replay. However, while AQP is targeted towards sensor networks and prioritizes the cost of data acquisition, our system, FlorDB, emphasizes on the need for agile, retrospective, and evolving data logging and analysis within the context of machine learning. FlorDB introduces a specialized solution for the high-velocity ML experimentation process, focusing on adding and propagating logging statements to previous versions of code and efficiently managing and querying the resulting historical log data.

The utilization of code block alignment algorithms, such as GumTree [5], demonstrates a direct connection between the process of hindsight logging and well-established practices within the software engineering field. These algorithms offer a way to manage the inherent complexity of an evolving codebase, much in the same way that machine learning model development requires tracking of various iterations over time. Notably, the concept of "edit scripts" provides a mechanism for capturing the delta between different versions of code, and enables us to inject new logging statements between aligned code-block boundaries. This concept has found wide application in areas such as code patching and merging [6].

In conclusion, the practices and challenges surrounding Multiversion Hindsight Logging bear distinct parallels and intersections with numerous fields. Whether in relation to the systems literature, databases, or software engineering, the application of existing concepts, practices, and algorithmic solutions proves to be instrumental in enhancing the agility of machine learning experimentation. At the same time, it is clear that the unique demands of ML experimentation require specialized solutions, such as our proposed system, FlorDB, which further develops these ideas to address the specific challenges faced by Machine Learning Engineers.

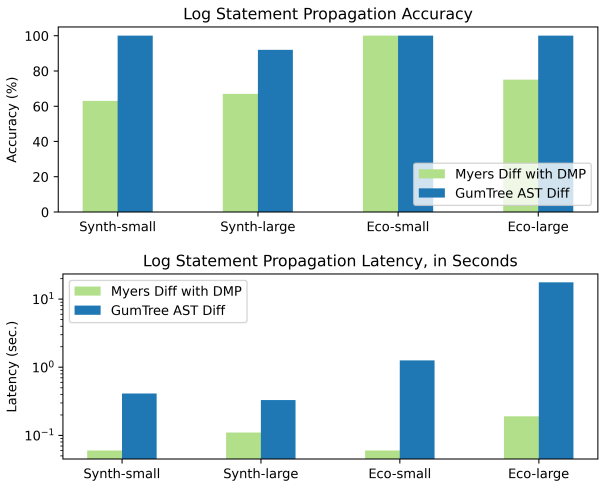


Figure 2: Accuracy and Latency for two alternative algorithms for logging statement propagation. Evaluation is over samples of syntehtic (Synth) and ecological (Eco) datasets.

3 ITERATIVE BATCH PROCESSING

FlorDB is designed to respond to the iterative and exploratory nature of the MLE workflow. It incorporates a suite of features that automate and expedite the process of checkpointing model training runs [8]. By implementing both background and adaptive checkpointing capabilities, FlorDB ensures that the overhead of iteration is minimized, promoting a swift and agile iteration loop.

FlorDB further bolsters the iterative MLE workflow through its integrated versioning support. Given that the model developer operates from a dedicated "shadow" branch—where automatic Git commits are allowed—FlorDB commits a snapshot of the project directory to Git after each execution. This feature enables detailed tracking of each step throughout experimentation, and enhances the reproducibility of the iterative MLE workflow.

To support grid search, or the execution of model training jobs by the batch, FlorDB employs a background server process grounded in the Worker-Pool execution model. Whether working within an interactive environment or a Jupyter Notebook, model developers can define and schedule a batch (or table) of experiments to be executed. These tasks are then systematically performed by the background server processes. Server processes may be added or removed on-the-fly as needed or as GPUs become available. This execution model promotes scalability, facilitates efficient workload management, and ensures a seamless and streamlined workflow, in line with the iterative and experimental nature of ML Engineering.

4 LOGGING STATEMENT PROPAGATION

Hindsight Logging represents an optimistic logging procedure, where an initial evaluation of the model is conducted followed by a subsequent identification of pertinent variables for tracking and logging. Expanding this concept over temporal dimensions, i.e., across different versions, elucidates the principle of Multiversion Hindsight Logging. The primary tenet of this approach is that any logging statement introduced in the most recent version of the model training code should also support historical queries by its extension or application to preceding versions, a phenomenon referred to as "logging statement propagation."

To support historical queries over variables logged in hindsight, FlorDB automatically propagates logging statements across previous versions. A non-trivial step in logging statement propagation involves the use of code version diffs to pinpoint the insertion site of the logging statement. In accomplishing this procedure, we employ a variant of the two-step GumTree diffing algorithm, which aids in the identification of semantically equivalent code blocks across different versions [5]. The GumTree diffing algorithm consists of the following two parts:

- **Greedy Top-down**: This stage seeks to identify all isomorphic subtrees within the Abstract Syntax Tree (AST) pair. These subtrees are sorted in descending order based on their heights, and "anchor mappings" are subsequently computed between the nodes of these isomorphic subtrees.
- **Bottom-up**: Upon the creation of anchor mappings, this phase constructs "container mappings." Here, two nodes are deemed to match if a substantial number of common anchor mappings are included among their descendants. If a match is confirmed, the algorithm commences a search for additional mappings, referred to as "recovery mappings", among the descendants during post-processing. The aggregate of these mappings is leveraged for code alignment. The post-processing stage additionally provides robustness to code refactoring and variable renaming.

The GumTree algorithm operates by seeking a suitably compact edit script to recognize equivalent code blocks. An edit script, in this context, encodes the difference between the target and source scripts using a set of operations such as `UpdateValue`, `Add`, `Delete`, and `Move`. The objective is for this edit script to encapsulate higher-order semantic actions, such as *move* and *update* operations, rather

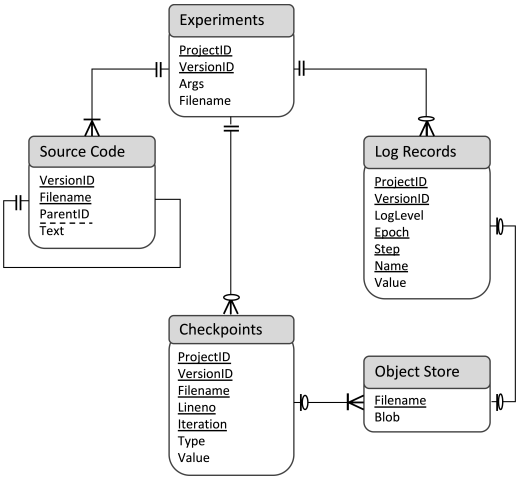


Figure 3: Data model diagram. Cardinality estimates are denoted in Crow's Foot notation.

than just syntactic or lexical modifications, such as line addition or deletion. Consequently, the manipulation of the Abstract Syntax Tree (AST) is integral to the algorithm's operation.

In a detailed technical report, we delve into the specifics of our Python implementation of the GumTree diffing algorithm, accompanied by a quantitative assessment of its accuracy and runtime in comparison with other leading code-diffing algorithms. The choice of GumTree is motivated by its superior performance with respect to both accuracy and runtime, as corroborated by the data presented in Figure 2.

5 RELATIONAL MODEL

FlorDB efficiently manages code, logs, and checkpoints in the Machine Learning Engineering (MLE) workflow. It utilizes Git for version control, associating metadata and storing important logs within the repository. To prevent overloading the version control system, large binary files are stored in a shared drive. FlorDB leverages SQLite to maintain a materialized view of log records, allowing for easy comparison and analysis of variables across different code versions. Multiversion Hindsight Logging in FlorDB enables the addition of rows and columns to the pivot table, expanding data collection and facilitating analysis.

FlorDB handles queries straightforwardly using SQLite, but for queries involving backfilling null values or incorporating hindsight logging, it employs Flor Replay operators. These operators align the Git repository to the desired version and perform backfilling, ensuring data consistency and completeness. In the replay mechanism, three key operators—DP, DP^*, and DP_i—execute the model training code with different levels of memoization and parallelism. FlorDB's replay mechanism estimates time requirements based on initial measurements, and calibration is performed when replays occur on different machines to adjust time estimates accurately.

Overall, FlorDB effectively manages code, logs, and checkpoints, provides a comprehensive view of variables across code versions, and offers efficient null value backfilling through its replay mechanism.

5.1 Data Model

FlorDB, our proposed system, is responsible for managing code, logs, and checkpoints. It operates under the assumption that it constitutes one among many systems in the MLE stack. Given the widespread use of data versioning by Machine Learning Engineers, FlorDB manages data by reference only, optimizing storage and operational efficiency. FlorDB leverages Git for version controlling code, committing code upon every execution. Concurrently, metadata pertinent to each commit, including version identifiers, are associated with Flor.

FlorDB serves as an integrative system that not only links metadata but also stores crucial logs, such as loss and accuracy metrics, within the Git repository. This design is intentional, promoting an architecture where access to the model training code (i.e., the repository) inherently includes access to a historical record of past executions. This coupling of code and logs enhances traceability and reproducibility in the machine learning workflow.

Nevertheless, certain exceptions are made in the case of logs (and checkpoints) pointing to large binary blobs. In such instances, while their references are maintained within the Git repository, the binary file is stored in a shared drive. This design choice is made to avoid overburdening the version control system, ensuring the continued efficiency of version synchronization operations. It also addresses concerns related to storage capacity in the Git repository, ensuring the long-term viability and scalability of the system.

To summarize, FlorDB efficiently manages code, logs, and checkpoints, relying on Git for version control. Each code execution is followed by a commit, with all associated metadata, including version identifiers. The system integrates metadata and important logs like loss and accuracy metrics within the Git repository, linking access to model training code and historical execution records, thus bolstering traceability and reproducibility in machine learning workflows. Exceptions are made for large binary blob logs, where references are kept in the Git repository, but the file is stored in a shared drive to preserve version synchronization efficiency and storage capacity. For a complete view of FlorDB's data model, refer to the schema diagram in Figure 3.

5.1.1 The Pivoted Views. FlorDB, capitalizing on the lightweight and portable features of SQLite, maintains a materialized view of log records preserved within the Git repository (Figures 4 & 5). The SQLite file is stored in the shared drive along the binary blobs and checkpoints. Additionally, FlorDB dynamically generates a pivoted view at analysis time, which enhances the visibility of logged variables and supports feature-based conditioning or filtering. In this architecture, each log record finds correspondence with a single cell within the pivoted view. The pivot operation is performed on the dimensions of `name` and `value`, thereby transforming the multi-dimensional log data into a more manageable two-dimensional table. Under this schema, every variable logged by Flor emerges as a unique column within this table, whereas each row encapsulates information pertaining to a distinct iteration instance or code version. This structure has two primary benefits: It offers an comprehensive view of all variables across varying code versions, thereby facilitating comparative and trend analyses. Simultaneously, it supports effortless conditioning or filtering based on distinct variables,

```
WITH data_prep AS (
   SELECT CONCAT(projectid, '.',
      versionid) rowid, name, val
   FROM log_records
   WHERE loglevel = 'd')
CREATE VIEW data_prep_pivot AS
SELECT * FROM crosstab(
   (SELECT * FROM data_prep
      ORDER BY rowid, name),
   (SELECT DISTINCT name
      FROM data_prep));
```

```
WITH outer_loop AS (
   SELECT CONCAT(projectid, '.',
      versionid, '.', epoch) rowid,
      name, val
   FROM log_records
   WHERE loglevel = 'o')
CREATE VIEW outer_loop_pivot AS
SELECT * FROM crosstab(
   (SELECT * FROM outer_loop
      ORDER BY rowid, name),
   (SELECT DISTINCT name
      FROM outer_loop));
```

```
WITH inner_loop AS (
   SELECT CONCAT(projectid, '.',
      versionid, '.', epoch, '.',
      step) rowid, name, val
   FROM log_records
   WHERE loglevel = 'i')
CREATE VIEW inner_loop_pivot AS
SELECT * FROM crosstab(
   (SELECT * FROM inner_loop
      ORDER BY rowid, name),
   (SELECT DISTINCT name
      FROM inner_loop));
```

Figure 4: Pivoted Views over Log Records.

```
CREATE VIEW full_pivot AS
SELECT * FROM data_prep_pivot
   FULL JOIN outer_loop_pivot
      USING (projectid, versionid)
   FULL JOIN inner_loop_pivot
      USING (projectid, versionid, epoch)
ORDER BY projectid, versionid, epoch, step;
```

Figure 5: Full Pivot View

significantly simplifying exploratory data analysis and hypothesis testing.

5.2 Queries

Multiversion Hindsight Logging, in essence, can be reduced to operations involving queries and modifications on the full pivot table. These modifications include not only the addition of rows, which aligns with the execution of additional experiments, but also the capacity to add columns and backfill null values.

The process of adding columns is accomplished by incorporating additional hindsight logging statements into the model training code. This enhancement extends the depth of information collected and reflected in the pivot table, providing new avenues for data analysis and comparison across code versions and executions.

Backfilling null values is another significant feature of this approach. Null values can appear in the pivot table when new columns are added, representing logged variables that did not exist in earlier executions. Backfilling these nulls is accomplished through issuing a specific query to FlorDB and executing a 'Flor replay.' This process allows for the generation of these missing values based on the current and past states of the model, providing a more comprehensive and consistent view of the data.

5.2.1 Adding Virtual Columns. FlorDB enriches its pivot table through logging statements, where the addition of columns is facilitated by the integration of extra hindsight logging statements into the model training code. This strategy bolsters the depth of the captured information, expanding the horizon for data analysis and comparison across various code versions and executions.

The addition of a logging statement to the code essentially gives rise to a corresponding virtual column on the full pivot table. Following a swift static scan of the file by the Flor system, the appearance of a new virtual column, labeled as 'version', becomes evident in the full pivot view. However, this virtual column is initially vacant, awaiting population with relevant data during subsequent executions.

5.2.2 Backfilling Nulls. The process of backfilling nulls in FlorDB's pivot table, handled by Machine Learning Engineers (MLEs), begins with a query submission to select specific versions and, as necessary, associated epochs and iterations. FlorDB identifies the required versions and iterations of model training, after which relevant logging statements are propagated to all selected versions. This inclusion guarantees accurate reflection of operations performed during these specific iterations and versions. It may be necessary to propagate logging statements even without new hindsight logging statements. This is because different experiments log unique variables, and certain logging statements might be excluded if they introduce too much overhead or impair code readability.

After normalizing all chosen versions to include the correct set of logging statements, a selective replay with Flor is executed. This replay process involves rerunning specific sections of training associated with the null values in the logs, filling the blanks based on the model's current and historical states. Upon the completion of Flor Replay, all selected nulls within the pivot table are effectively backfilled, ensuring a comprehensive and consistent representation of data.

5.3 Calibrated Physical Operators

Queries that do not involve hindsight logging, or backfilling nulls, are simple and managed solely by the database management system, which in the context of FlorDB is an instance of SQLite. However, in queries where backfilling nulls is required, FlorDB invokes Flor Replay operators.

FlorDB first aligns the Git repository hosting the model training code to the appropriate version. This alignment is followed by the execution of Flor Replay operators, which are parameterized by two main aspects: replay mode and work partitioning. The focus of this discussion is the parameterized Flor Replay operator, a critical component in backfilling nulls and maintaining data consistency and completeness in FlorDB's pivot table representation.

5.3.1 Physical Query Operators. In the scope of machine learning diagnostics, feature values such as loss or accuracy can serve as query predicates for other features like tensor histograms. This

Table 2: Computer vision and NLP benchmarks used in our evaluation.

Model	Model Size	Data	Data Size	Objective	Evaluation	Application
ResNet-152 [11]	242 MB	ImageNet-1k [17]	156 GB	image classification	accuracy	computer vision
BERT [4]	440 MB	Wikipedia [7]	40.8 GB	masked language modeling	accuracy	natural language processing
GPT-2 [16]	548 MB	Wikipedia [7]	40.8 GB	text generation	perplexity	natural language processing
LayoutLMv3 [13]	501 MB	FUNSD [10]	36 MB	form understanding	F1-score	document intelligence
DETR [1]	167 MB	CPPE-5 [2]	234 MB	object detection	μ-precision	computer vision
SegFormer [20]	14.4 MB	Sidewalk	324 MB	semantic segmentation	μ-IOU	computer vision
TAPAS [12]	443 MB	WTQ [15]	429 MB	table question answering	accuracy	document intelligence

functionality allows developers to navigate through the training history using these metrics to pinpoint anomalies or areas of interest. In this context, an MLE may formulate a SQL query to select specific versions of the code, based on parameters such as training characteristics (e.g., batch size, learning rate) or performance metrics (e.g., loss, accuracy). To facilitate detailed analysis and efficient backfilling of null values in the pivot table, FlorDB offers a replay mechanism featuring various operators:

- *DP*: As the default replay operator, *DP* executes the entire model training code, traversing the main epoch loop but bypassing the nested training loop by loading checkpoints. This operator is particularly useful when hindsight logging statements are incorporated into the model validation section of the code.
- DP^*: When parameterized for model evaluation, this replay operator also executes the complete model training code but skips the main loop by loading the final checkpoint from training. Flor resorts to this operator when hindsight logging statements are added into the model evaluation or testing section of the code, which typically follows the main loop.
- DP_i: This operator can be parameterized by the number of workers (n), and work partition (i). It holds value for parallelizing model training replay, as well as for selective replay. For instance, if the model developer wishes to observe the tensor histograms at a specific iteration, FlorDB can adjust the number of workers to match the number of epochs, and step into the desired iteration. Flor employs this replay operator for the most granular logging: whenever model developers introduce hindsight logging statements into the nested training loop.

A unique advantage of FlorDB's replay mechanism lies in its ability to leverage time measurements from the initial run to accurately estimate the time requirements of subsequent replays. This feature addresses the longstanding challenge of cost estimation in the realm of databases. In traditional databases, accurate cost estimation requires a deep understanding of data distribution and query processing mechanisms. However, FlorDB simplifies this process by harnessing data from the initial model training execution to generate precise cost predictions for future replays.

This estimation process may require calibration when the training replay occurs on a different machine or GPU than the original. This scenario necessitates a calibration step that measures the new machine's IO read/write specifications, enabling FlorDB to adjust time estimates accordingly. While the accuracy of these estimates might vary when switching between systems due to differences in system characteristics, Flor's calibrated approach provides reasonably close estimates, thereby ensuring a level of predictability in system behavior.

6 EVALUATION

[JMH: Of the Multiversion Hindsight Logging Query Engine.] We evaluated flordb across 8 different model architectures (Table 2). In the table, we show how model architectures vary by model size, as well as the kind and size of training data that they encode. We wish to show that flordb is useful for iterative ml, and show how it generalizes to all areas of ai/ml: computer vision, nlp, document intelligence, and reinforcement learning. In flor, we demonstrated that flor achieved efficient record and parallel replay. Here, we show how flor scales up and out across versions.

First, we show that flordb generalizes flor to scale-up to replaying hundreds of versions (as opposed to just one): that despite executing queries that access hundreds of versions, that the response times (Figure 6) and storage footprint (Figure 7) are acceptable.

Moreover, we demonstrate that model developer time and effort is not increased by generalizing flordb across versions: after logging a single version (e.g. latest), model developers can automatically propagate that logging statement to all other versions using software patching techniques with high accuracy (Figure 2), meeting the requirement that the model developer not have to manually log hundreds of versions – a non-starter.

Finally, we show that when faster response times are needed, e.g. for interactive analytics and model diagnostics sessions, flor can provide accurate cost estimates and steer the user toward more selective queries (Figure 8), and scale-out to dozens of GPUs to exploit the embarrassingly parallel nature of cross-version replay and within-version parallelism from checkpoint-resume (Figure 9).

6.1 Logging statement propagation is accurate

We will evaluate the **accuracy** of statement propagation and statement adaptation. A technical report evaluates accuracy of several statement propagation strategies using hand-constructed samples, a group of randomly generated variations of PyTorch programs, and a set of popular code bases [3].

May be sensitive to control flow changes, but GumTree algorithm should be robust to this. Can GumTree notify us if it doesn't know what to do, if it needs help, or believes there is higher risk of an error?

The evaluation relies on synthetic and ecological benchmarks.

- hand-constructed test examples

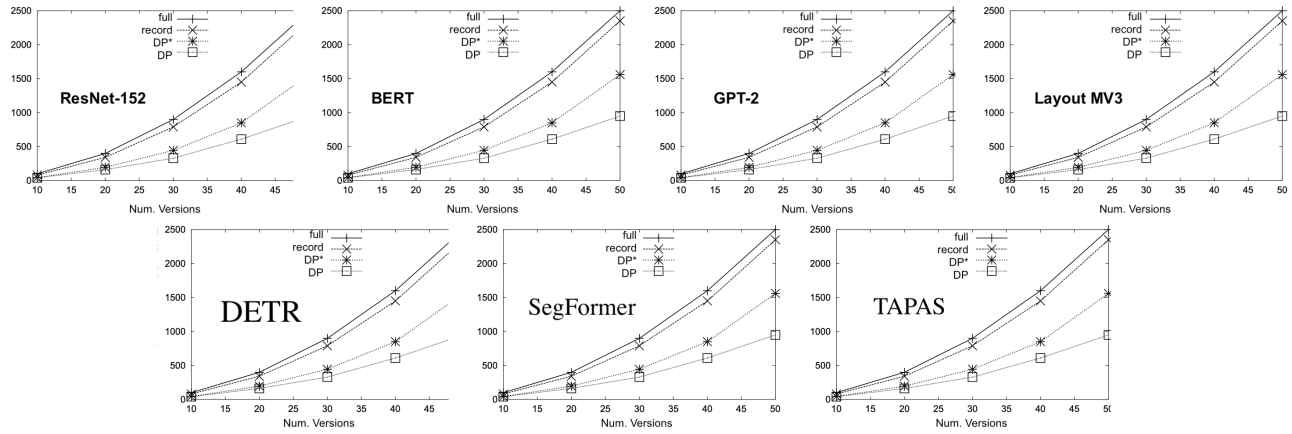


Figure 6: Grid of Query Replay times across versions

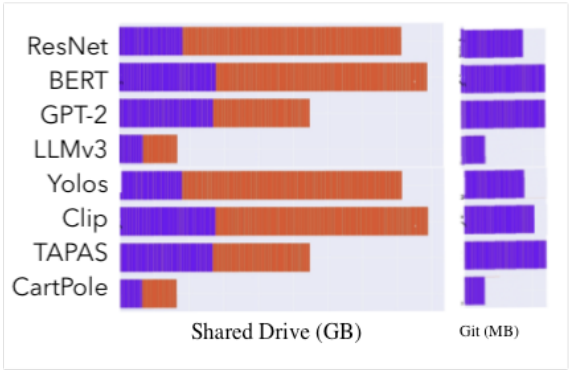


Figure 7: Storage footprint

- group of randomly generatd variations of PyTorch programs.
- A set of popular code bases.

6.2 Flordb scales up to hundreds of versions

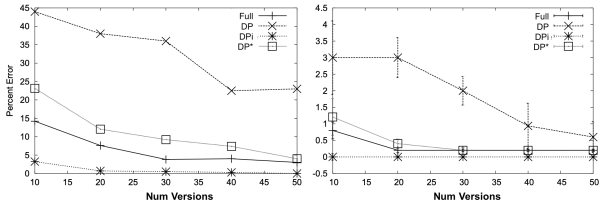
First, we show that flordb generalizes flor to scale-up to replaying hundreds of versions (as opposed to just one): that despite executing queries that access hundreds of versions, that the response times (Figure 6) and storage footprint (Figure 7) are acceptable.

6.3 Cost Estimates: Max and Mean errors

Finally, we show that when faster response times are needed, e.g. for interactive analytics and model diagnostics sessions, flor can provide accurate cost estimates and steer the user toward more selective queries (Figure 8), and scale-out to dozens of GPUs to exploit the embarrassingly parallel nature of cross-version replay and within-version parallelism from checkpoint-resume (Figure 9).

6.4 Flordb scales out from 0 to many GPUs

Finally, we show that when faster response times are needed, e.g. for interactive analytics and model diagnostics sessions, flor can



(a) Maximum Error (b) Average Error

Figure 8: Max and Mean errors for calibrated cost estimate.

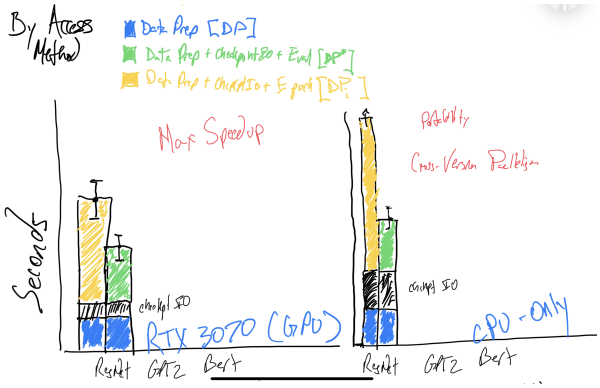


Figure 9: Stacked Bar Chart. Max-Speedup, I will want a further plot, where I report the

provide accurate cost estimates and steer the user toward more selective queries (Figure 8), and scale-out to dozens of GPUs to exploit the embarrassingly parallel nature of cross-version replay and within-version parallelism from checkpoint-resume (Figure 9).

7 CONCLUSION & FUTURE WORK

- Advanced replay interpolation: pluggable and extensible, advanced techniques.

- Replay priority: deliver tuples in priority order.
- Automatic Featurization: Some execution data is logged with such high frequency and regularity that it is sensible to capture and featurize it automatiaclly. Some tech companies, such as CometML and MLFlow, offer support for automatic featurization. At least, training hyper-parameters and metrics should be featurized automatically. One naive approach for automatic featurization is to log all the scalar values in a training run, and use the Python variable name for the feature name.

REFERENCES

[1] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-End Object Detection with Transformers. arXiv:2005.12872 [cs.CV]

[2] Rishit Dagli and Ali Mustufa Shaikh. 2021. CPPE-5: Medical Personal Protective Equipment Dataset. arXiv:2112.09569 [cs.CV]

[3] Anusha Dandamudi. 2021. *Fast Low-Overhead Logging Extending Time.* Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-117.html

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[6] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007), 725–743.

[7] Wikimedia Foundation. [n.d.]. *Wikimedia Downloads.* https://dumps.wikimedia.org

[8] Rolando Garcia, Eric Liu, Vikram Sreekanti, Bobby Yan, Anusha Dandamudi, Joseph E Gonzalez, Joseph M Hellerstein, and Koushik Sen. 2020. Hindsight logging for model training. *Proceedings of the VLDB Endowment* 14, 4 (2020), 682–693.

[9] Rolando Garcia, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E Gonzalez, and Joseph M Hellerstein. 2018. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, Vol. 114.

[10] Jean-Philippe Thiran Guillaume Jaume, Hazim Kemal Ekenel. 2019. FUNSD: A Dataset for Form Understanding in Noisy Scanned Documents. In *Accepted to ICDAR-OST*.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4320–4333. https://doi.org/10.18653/v1/2020.acl-main.398

[13] Yupan Huang, Tengchao Lv, Lei Cui, Yutong Lu, and Furu Wei. 2022. LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking. In *Proceedings of the 30th ACM International Conference on Multimedia*.

[14] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

[15] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. arXiv:1508.00305 [cs.CL]

[16] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[18] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. 2022. Operationalizing machine learning: An interview study. *arXiv preprint arXiv:2209.09125* (2022).

[19] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–3.

[20] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. 2021. SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers. arXiv:2105.15203 [cs.CV]

[21] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *Proceedings of the 2021 International Conference on Management of Data*. 2639–2652.

[22] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.