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

Koushik Sen
UC Berkeley
ksen@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

PVLDB Reference Format:
Rolando Garcia, Anusha Dandamudi, Gabriel Matute, Lehan Wan, Joseph Gonzalez, Koushik Sen, and Joseph M. Hellerstein. 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 behing 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" [23]. Sustaining high velocity over the course of model development and tuning leads to an explosion in the number of model versions. As businesses 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 [10, 25].

1.1 Modern ML: High Velocity Experimentation

Because experimentation is such an important part of model development, there are now many mature tools available for managing ML experiments, such as, TensorBoard [11], MLFlow [26], and WandB [1] — to name three popular examples. All of these tools expose a logging interface through a language library by which the tool collects runtime data and metrics from model training runs. Experiment management tools then i) organize the data logged by project name, version number, timestamp, and so on, and ii) show the logged data back to the user via tables and plots. At every step of exploration, model developers routinely log training data to evaluate past runs, debug training problems, and select subsequent experiments. The following are common examples of the kinds of training data logged by model developers

In sum, model developers will log parameters & metrics by default, and this data will be entered by experiment management tools into a table of experiments and visualized as timeseries plots inside a dashboard. Given this information, the model developer will then decide whether to log additional data, like tensors & embeddings or images & overlays, and as needed, they will then add logging statements to their model training code, and re-run.

[JMH: Agility. One paragraph on experimental agility in ML.]

1.2 Multiversion Hindsight Logging

As models and training data become larger and more complex, training times continue to grow from days to weeks. In this context, naïve re-executions of model training just for better logging can feel especially painful. In past work, we have looked into accelerating model training replay for hindsight logging through memoization and parallelism, using automatically-generated checkpoints in a system we call FLOR, for Fast Low-Overhead Recovery [9].

[JMH: Hindsight logging. One paragraph on Hindsight Logging 2020. Connect to agility, explain the user-facing functionality as a black box, and yes, Flor made it quick and cheap.]

[JMH: The problem: One paragraph on the problem we tackle here: Multiversion Hindsight Logging. Turns out, people want to

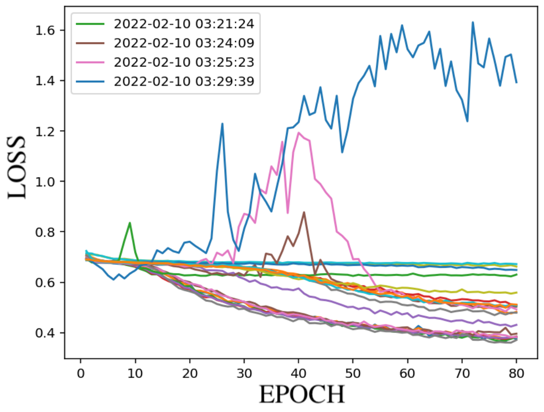


Figure 1: Loss landscape showing the many model versions that are normally explored as a result of model training and fine-tuning. Versions are seemingly similar, splitting into approximately three clusters and a handful of anomalies.

go back in time not just to the last run, but to prior runs on prior versions of training – different code, different data. We need a new system, System X, that builds on Flor but goes much further.]

1.2.1 Scenario. Maybe reconstruct recent (coredata) demo. Add screenshots. Of Jupyter, of tables, of source code, of confusion matrix. Alice just finished an exploration consisting of hundreds of versions. She would like to i) explain her exploration steps or history to the kaggle community or for a blog post, ii) create a reproducible notebook or docker container for the community to run with the push of a button. Scenario: handing off work for someone else to continue.

[JMH: Motivating Scenario: One paragraph scenario motivating some search on history ("find the version when the bounding boxes started to go crazy"). (Could be in a figure if it takes more than one paragraph.)] As the developer explores alternatives to tune a model, they may discover they need to start tracking additional metrics or intermediate values (e.g. tensor histograms or segmentation masks) beyond what they had been logging so far. In this case, the model developer will want to know the *historical* values for the metric or intermediate that is now of interest, and view how it changes over time. Naively, the model developer may recover historical values by revisiting past versions of code, adding the relevant logging statements, and replaying model training from checkpoints. But this manual intervention is impractical for more than a small handful of versions. Instead, the model developer would like to add logging statements to their latest version of code, and automatically apply the patch to historical versions. In a sense, this is like rewinding time to insert the desired logging statements *from the beginning* of history. We refer to this practice, of augmenting or backfilling existing charts or tables with values from historical versions, as hindsight logging.

Three Key Challenges:

(1) if user adds log statement to version n, how do we propagate to prior versions n-k with different code,
(2) how do we allow users to deal with all this history by asking hindsight logging "queries" over many past versions, and
(3) given there are infinitely many possible log statements and lots of versions, how does System X execute the "queries" efficiently, and how do we help users avoid wasting too much time and GPU budget?

1.3 Contributions

[JMH: Our approach: FlorDB is a Multiversion Hindsight Logging system with a familiar SQL-like query interface. Each log statement ever added is represented as a "virtual column"; each run of a version is captured as a row.] In addition to completing the relevant engineering work for automatic version control, and achieving feature parity with state-of-the-art tools for experiment management of machine learning, we claim the following research contributions:

[JMH: Our approach: Topic (1) has been studied in the SE literature for XX purposes. Since our workload is specialized differently, we evaluate past ideas in this context, and find that GumTree is good.]

[JMH: Our approach: For (2), we introduce a "virtual table" view of the history of our logs, and allow users to ask their questions in familiar SQL (or pandas). Logging becomes a metaphor of "adding a virtual column" (corresponding to a new log statement) and then Hindsight Logging can backfill that column by efficiently recomputing the log outputs across many versions.]

[JMH: Our approach: For (3) we focus on two issues. (a) we define the "physical operators" for replay queries, show how they perform, and how they compose, and (b) we show how we provide highly-accurate wall-cost estimates for the operators to help with the user experience – can be surfaced in estimates, progress bars, etc.]

2 BACKGROUND ON HINDSIGHT LOGGING

[JMH: Maybe we do related work here too.] We will conduct a review of the literature in PL and databases.

- Prior work into model version control: model repositories, DAG or graphviz diagrams, metrics dashboards, etc. Code-centric view of version control. Counter-cultural rallying cry: git gets it right.
- Daikon invariant detection [6, 16–18, 20] . . . and more!
- Type theory and type systems.

3 LOGGING STATEMENT PROPAGATION

We use a variant of the two-step GumTree diffing algorithm to identify semantically equivalent code blocks across versions [7].

- Greedy Top-down: finds all isomorphic subtrees in AST pair. Sorts subtrees by decreasing height. Adds "anchor mappings" between the nodes of the isomorphic subtrees.
- Bottom-up: Creates container mappings based on anchor mappings: two nodes match if their descendants include a large number of common anchor mappings. When two nodes match, the algorithm searches for additional mappings among the descendants in post-processing, and this is called "recovery mappings". The aggregation of mappings

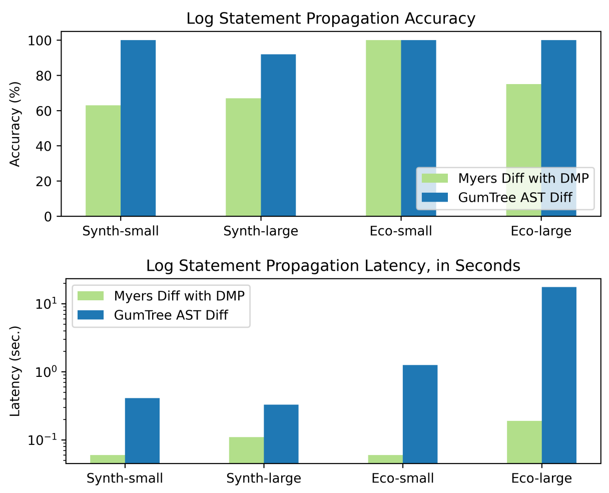


Figure 2: Accuracy and Latency for two alternative algorithms for logging statement propagation. Evaluation is over samples of syntehtic (Synth) and ecological (Eco) datasets.

is used for code alignment. Post-processing: code refactoring and variable renaming tolerance based on syntactic structure.

Formally, the problem of log propagation involves finding a "small enough" (can't say minimal) edit script. An edit script is the difference between the target script and the source script, and is composed of `UpdateValue`, `Add`, `Delete`, and `Move` operations.

We want the edit script to describe higher-order semantic actions, like move and update, not just syntactic or lexical modifications like add or delete line. Thus, the algorithm should use the AST.

The best known algorithm with add, delete, and update actions has cubic time complexity. Computing the minimum edit script that can include move actions is known to be NP-hard. In our case, because of what training scripts look like in practice, optimal algorithms may be an option.

GumTree uses heuristics and pragmatic rules, these can be tailored to model training and Python specifically.

Edit scripts are deduced after first inferring mappings between nodes in the source AST and nodes in the target AST.

The patching library open source and available at:
github.com/ucbrise/hlast.

4 RELATIONAL MODEL

For multiversion hindsight logging.

4.1 Data Model

In machine learning, logging statements are used to featurize values from training runs. Model developers may choose to log training hyper-parameters, intermediate execution data, or metrics (e.g. the loss or accuracy) with specialized tracking and visualization libraries such as TensorBoard [11] and WandB [1]. In order to extract a value from a training execution and transform it into a proper feature for query or visualization, logging libraries for ML will capture execution metadata for the value at runtime, such as its name or the epoch when it appeared.

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.

Table 2: Table Definitions

Table	Sym.	Definition
Experiments	$\mathbb{E}$	$\langle p, v, a, n \rangle$
Source_Code	$\mathbb{G}$	$\langle v, n, \hat{v}, t \rangle$
Checkpoints	$\mathbb{C}$	$\langle p, v, n, l, i, y, x \rangle$
Object_Store	$\mathbb{S}$	$\langle n, b \rangle$
Log_Records	$\mathbb{F}$	$\langle p, v, g, e, s, m, x \rangle$

Henceforth, we use $\mathbb{G}$ to refer symbolically to a Git repository, and formally refer to it as follows:

We will want to include some details about how we commit to Git on every run of model training, and the shadow branches used. Source code is part of the data model because it helps filter different executions of the same script, and storing the source code is necessary to enable replay, part of hindsight logging.

This is where we discuss the checkpoints table and associated object store.

Add a paragraph on **buffer management** in the cloud, S3, and LRU.

Henceforth, we use $\mathbb{F}$ to refer symbolically to a training feature, and define it as follows:

The reader should note that each value is uniquely identified by the conjunction of the projectID, versionID, epoch, step, and name. We assume that any value—intensional or extensional (i.e. derived)—of a model training execution can be featurized. Filename is omitted because the name of the logged variable is globally unique, so there

```
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 3: 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 4: Full Pivot View

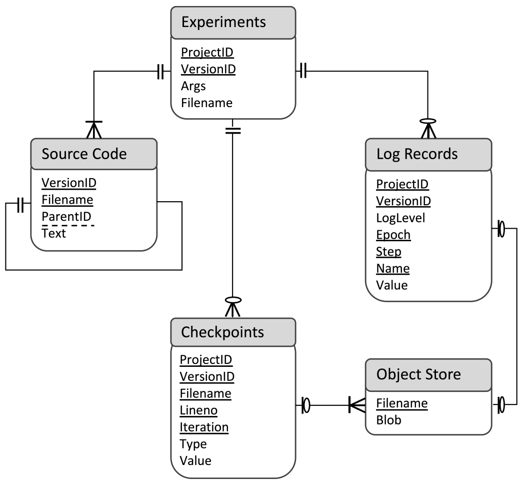


Figure 5: Data model diagram. Cardinality estimates are denoted in Crow's Foot notation.

is no ambiguity. We will also add a note about how we deal with large objects by reference, and a quick note about serialization.

We can refer to a figure showing what a plot for a loss would look like in a TensorBoard dashboard. For illustration, we will show i) source code with statement logging the loss, ii) a table with many entries for the loss, iii) and a plot of the loss curve side-by-side. The plot (iii) is not part of the data model, but a view over the fact table.

Talk about Git and Version Control.

4.1.1 The Pivoted Views. First we motivate why we want a pivot table, i.e. ease of understanding, ability to conditione or filter on other features. The pivot table is a view over logging with the following properties. Queries are possed in SQL over the pivot table of some experiment. FLOR rewrites the query into a set of queries over the logging table. In this sense, the pivot table is a SQL view over logging. Define the pivot table with SQL.

And then we take each of the three pivoted views, and full-join them on overlapping dimensions to produce a full pivoted view:

4.2 Queries

There is one relational table per model training project (e.g. traffic monitoring, English-French translation), every such table has columns `version_id`, `epoch`, and `step` corresponding the the *dimensions*, or address, of features. Additionally, each table has as many columns, corresponding to *measures*, as there are named logging statements in the model training code. Some examples are losses, tensor histograms, segmentation masks, and so on.

4.2.1 Adding Virtual Columns. Adding a logging statement to the code creates a virtual column on the FULL PIVOT table.

After a fast static scan of the file by Flor, we can see that a new virtual column for version has been added to the FULL PIVOT view. The virtual column is empty.

```python
import torch
import flor
from flor import MIK as Flor

Flor.checkpoints(net, optimizer)
for epoch in Flor.loop(range(...)):
    for data in Flor.loop(trainloader):
        optimizer.zero_grad()
        inputs, labels = data
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    eval(net, testloader)
```

Figure 6: Training script with Flor checkpointing.

Table 3: Computer vision and NLP benchmarks used in our evaluation.

Model	Model Size	Data	Data Size	Objective	Evaluation	Application
ResNet-152 [13]	242 MB	ImageNet-1k [22]	156 GB	image classification	accuracy	computer vision
BERT [5]	440 MB	Wikipedia [8]	40.8 GB	masked language modeling	accuracy	natural language processing
GPT-2 [21]	548 MB	Wikipedia [8]	40.8 GB	text generation	perplexity	natural language processing
LayoutLMv3 [15]	501 MB	FUNSD [12]	36 MB	form understanding	F1-score	document intelligence
DETR [2]	167 MB	CPPE-5 [3]	234 MB	object detection	μ-precision	computer vision
SegFormer [24]	14.4 MB	Sidewalk	324 MB	semantic segmentation	μ-IOU	computer vision
TAPAS [14]	443 MB	WTQ [19]	429 MB	table question answering	accuracy	document intelligence

4.2.2 *Backfilling Nulls.* Log statement propagation is the first step. You may need to propagate logging statements even if you didn't add any hindsight logging statements, since different experiments log different things, and log statements that were added can be dropped if they add too much overhead or hurt readability.

4.3 Calibrated Physical Operators

By now, the reader will understand that featurization revolves around logging statements. Model developers featurize training runs with logging statements. Statement propagation is a system of techniques for patching historical versions of Python source code, so that past versions include the full set of applicable logging statements defined on the latest version, and are patched as necessary to preserve the intended semantics. More details on this work are available in a technical report [4].

4.3.1 *Physical Query Operators.* We note that the values of some features (e.g. loss or accuracy) can serve as query predicates for other features (e.g. tensor histograms). In a common diagnostics case, a model developer uses loss curves, or other metrics, to navigate the training history. Where there is evidence of an anomaly, model developers drill-down—with logging statements and SQL queries—and inspect more data with finer detail.

- DP: −replay_flor
- DP^*: −replay_flor 0/1
- DP_i: −replay_flor i/n

There is a large class of debugging queries that we can answer with one checkpoint, without any replay. These are queries that project away the `step` dimension and request information at no finer-grain than `epoch` level detail. More specifically, when we consider the training loss curve, there are regions of interest that show an anomaly. When this is the case, we can get away with replaying a small relevant range of training, and ignoring the rest (here, we reference an illustrative figure). Ultimately, selectivity is about relevant ranges and spans of interest. We should give special attention to **peak** and **end** epochs in line with behavioral experiments conducted by D. Kahneman.

Implementation Sketch

(1) Parse SQL Query
(2) Classify predicates and columns as either default features ($\mathcal{F}$) or hindsight features ($\widehat{\mathcal{F}}$)
(3) Map default feature values satisfying the query predicate to their respective dimensions $\{v, e, s \in \mathcal{F} | Pred(x \in \mathcal{F})\}$
(4) Use the predicate dimensions to generate a Flor replay plan, over a narrower replay space.
(5) Replay, enter data into table, and serve query

Replay avoidance is a special case of selectivity, but it merits separate treatment. Some data for featurization is either static or dynamic but invariant (or constant). Examples of static data are the hyper-parameter configurations. Examples of time-invariant dynamic data is the shape of the network layers. There are some techniques here we can explore around concolic or abstract interpretation, together with compiled binary instrumentation to side-step Flor replay. It may be possible to slash this portion from the paper and include it instead in future work.

4.3.2 *Performance Calibration.* Determining the replay schedule, data prep schedule, outer loop schedule, inner loop schedule. Versions selection from full pivot. estimating how long replay. How many GPUs are available for parallelism? Shortening running times with cost estimation, query selectivity.

Precise cost estimates. Make list of what's different here that we could not do in a traditional SQL env. Cost estimation for these functions is perfect. Cost of range query or point query relative to checkpoint can be measured exactly. You can imagine this as Query Processing problem, here's all the ways that's different.

5 EVALUATION

[JMH: Of the Multiversion Hindsight Logging Query Engine.] We evaluated flordb across 8 different model architectures (Table 3). 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 7) and storage footprint (Figure 8) 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

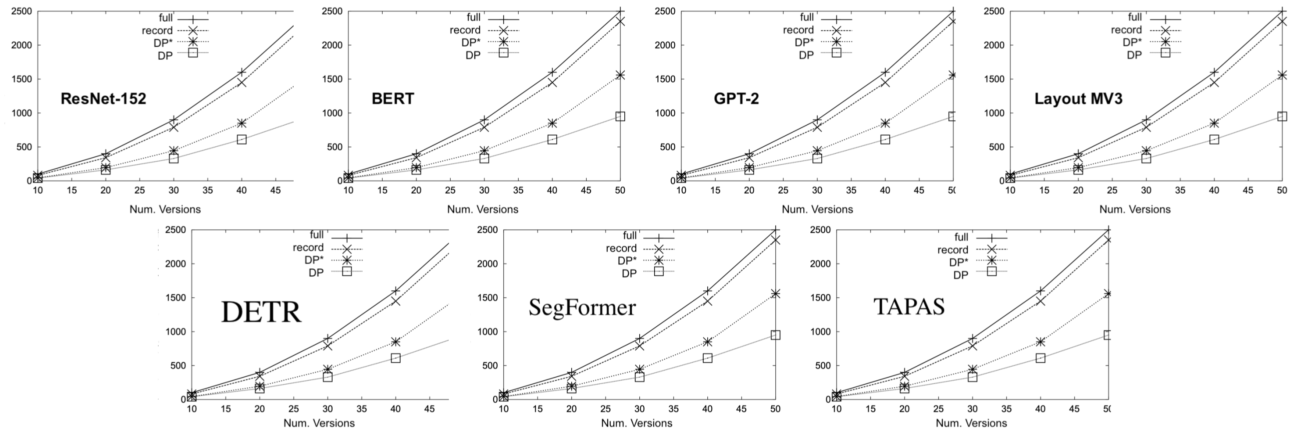


Figure 7: Grid of Query Replay times across versions

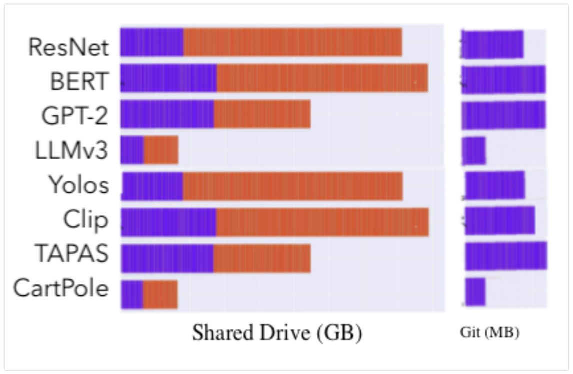
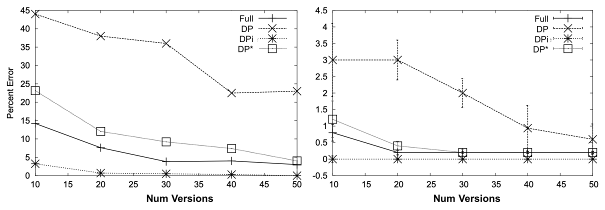


Figure 8: Storage footprint



(a) Maximum Error (b) Average Error

Figure 9: Max and Mean errors for calibrated cost estimate.

can provide accurate cost estimates and steer the user toward more selective queries (Figure 9), 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 10).

5.1 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 [4].

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

5.2 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 7) and storage footprint (Figure 8) are acceptable.

5.3 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 9), 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 10).

5.4 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 provide accurate cost estimates and steer the user toward more selective queries (Figure 9), 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 10).

6 CONCLUSION & FUTURE WORK

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

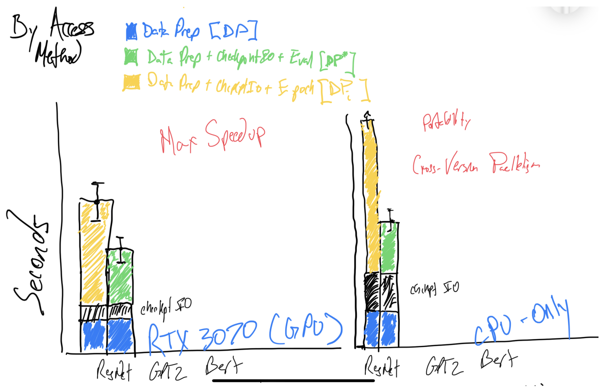


Figure 10: Stacked Bar Chart. Max-Speedup, I will want a further plot, where I report the

- 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] 2020. Weights & Biases. wandb.com.

[2] 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]

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

[4] 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

[5] 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

[6] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering* 27, 2 (2001), 99–123.

[7] 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.

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

[9] 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.

[10] 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.

[11] Google. 2020. TensorBoard. tensorflow.org/tensorboard.

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

[13] 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.

[14] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pretraining. 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

[15] 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.*

[16] Yoshio Kataoka, Michael D Ernst, William G Griswold, and David Notkin. 2001. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, 736–743.

[17] Jeremy W Nimmer and Michael D Ernst. 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science* 55, 2 (2001), 255–276.

[18] Jeremy W Nimmer and Michael D Ernst. 2002. Automatic generation of program specifications. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 229–239.

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

[20] Jeff H Perkins and Michael D Ernst. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. In *proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering.* 23–32.

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

[22] 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

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

[24] 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]

[25] 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.

[26] 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.