
# Kishu: Time-Traveling for Computational Notebooks

Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, Yongjoo Park
University of Illinois at Urbana-Champaign
{zl20,supawit2,ribhav2,assheth2,yongjoo}@illinois.edu

## ABSTRACT

Computational notebooks (e.g., Jupyter, Google Colab) are widely used by data scientists. A key feature of notebooks is the interactive computing model of iteratively executing *cells* (i.e., a set of statements) and observing the result (e.g., model or plot). Unfortunately, existing notebook systems do not offer *time-traveling to past states*: when the user executes a cell, the notebook *session state* consisting of user-defined variables can be *irreversibly modified*—e.g., the user cannot 'un-drop' a dataframe column. This is because, unlike DBMS, existing notebook systems do not keep track of the session state. Existing techniques for checkpointing and restoring session states, such as OS-level memory snapshot or application-level session dump, are insufficient: checkpointing can incur prohibitive storage costs and may fail, while restoration can only be inefficiently performed from scratch by fully loading checkpoint files.

In this paper, we introduce a new notebook system, Kishu, that offers time-traveling to and from arbitrary notebook states using an efficient and fault-tolerant incremental checkpoint and checkout mechanism. Kishu creates incremental checkpoints that are small and correctly preserve complex inter-variable dependencies at a novel *Co-variable* granularity. Then, to return to a previous state, Kishu accurately identifies the *state difference* between the current and target states to perform incremental checkout at sub-second latency with minimal data loading. Kishu is compatible with 146 object classes from popular data science libraries (e.g., Ray, Spark, PyTorch), and reduces checkpoint size and checkout time by up to 4.55× and 9.02×, respectively, on a variety of notebooks.

**PVLDB Reference Format:**
Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, Yongjoo Park. Kishu: Time-Traveling for Computational Notebooks. PVLDB, 18(4): 970 - 985, 2024.
doi:10.14778/3717755.3717759

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/illinoisdata/kishu-vldb.

## 1 INTRODUCTION

Computational notebooks (e.g., Jupyter [81, 144], Rstudio [121]) are widely used by data scientists [113, 114]. A key feature of the notebook workflow is iterative code execution and result observation [6, 25], which is highly compatible with the incremental nature of data science tasks, such as interactive tutorials [80], data exploration [37, 43, 166], visualization [45], and model tuning [19, 155]. This iterative workflow is enabled by notebooks systems being *stateful*—to do work, users would start a *session*, then as users execute code in the notebook system, the results are held in the *session state* as user-defined variables (e.g., loaded datasets, fitted models).

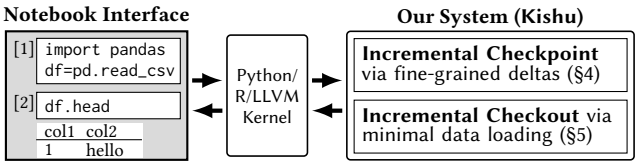


**Figure 1: Our system (attached to the kernel, right) enables time-traveling to and from arbitrary notebook states.**

***Limitation: no Time-Traveling for Notebooks.*** Oftentimes, during a workflow, users would like to revert changes made to the session state (i.e., *time-travel*), such as to undo a modification (e.g., restore a dropped column of a dataframe [140]), restoring an overwritten variable [72], or perform reverse debugging [23]. Unfortunately, unlike program debuggers (e.g., gdb) [16, 61, 115], relational databases (e.g., PITR in PostgreSQL and MySQL [70, 112]) or interactive data systems [37, 43, 87] which support time-traveling to past program states, existing notebook systems do not natively keep track of past session states: cell executions *cannot be undone*, e.g., the user cannot 'un-drop' a dataframe column. If the user executes a cell that alters the session state, a common approach to restore the previous state would be to restart the kernel and then (painstakingly) re-run past cells in the correct order. While code versions can be saved using tools such as Git [64] or native commands (e.g., Jupyter's %checkpoint [145][1]) to simplify identifying cells to rerun for restoration, cell reruns can still be time-consuming (e.g., re-training an ML model) and/or result in incorrect restoration (e.g., random train-test splits). Another approach is for the user to periodically checkpoint the session state (e.g., memory dump [7, 35] or session state serialization [58, 158]) to storage or a managed database (e.g., KV-store [146]). Then, users can load an appropriate checkpoint file to restore the session state. However, performing session checkpointing and restoration using these tools is limiting: checkpointing can incur prohibitive costs (§7.3, §7.4) and may fail on certain workloads (e.g., GPU [1]), and restoration can either (1) only be (inefficiently) performed *from scratch*, requiring completely loading a checkpoint file [58] and/or killing the current kernel [35], or (2) may be *incorrect*, breaking inter-variable relations [146].

***Our Goal: Generalizable, Correct, and Efficient Time-Traveling.*** We propose Kishu, a notebook system that enables time-traveling between session states: as the user executes cells, Kishu tracks the

[1]Despite its name, %checkpoint only stores cell code and not objects in the state.


Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717759

**Table 1: Comparison between Kishu and other possible approaches for time-traveling between session states**

| Approach | Mechanism |
|---|---|
| Session Migration tools [57, 94] | Replicates individual session states (no incremental checkpoint and restore) |
| OS-level Checkpoint tools [7, 35, 60, 79, 82] | Incrementally saves dirty memory pages (no incremental restore, fails with multiprocessing) |
| Notebook code versioning [22, 73, 145] | Versions notebook cell code and outputs for easier rerunning (but no directly restoring to a state) |
| PITR/time-travel in DBMS/HPC [8, 84, 95, 111, 129] | Incrementally checkpoints tables/KV-stores (We handle arbitrarily complex, interdependent objects) |
| **Ours (**Kishu**)** | **Efficient incremental checkpoint and checkout, generalizable to almost all session states** |

session state evolution while writing per-cell incremental *checkpoints* containing differing data between successive states (i.e., the *state delta*) for returning to any past state via an incremental *checkout* later. Kishu pursues three challenging goals—**Delta-Efficient Checkpoint:** Kishu aims to minimize incremental checkpointing overhead by exploiting the small per-cell deltas typical of data science workflows (§2.2), but also avoid high detection overhead in the face of complex access patterns and inter-variable dependencies. **Correct & Non-intrusive Checkout:** Kishu aims to restore past states in the same session non-intrusively by leveraging existing objects in the kernel (that don't need updating) to minimize data loading costs, while still guaranteeing checkout accuracy as if it completely loaded a checkpoint file. **Generalizability:** Kishu aims to support checkpointing/checkout for almost all notebook libraries (and/or use cases), of which there is a large variety, e.g., notebooks can perform distributed computing (e.g., Spark [165]) or move data off-CPU (e.g., GPUs [56]). If Kishu can achieve these goals, Kishu will allow users to undo almost any executed cell that undesirably modifies the state as if it never occurred by quickly checking out to the pre-execution state at the cost of minimal workflow overhead.

***Our Approach.*** Our core idea for achieving our goals is to capture the state delta with low overhead, but at sufficiently high granularity using information exclusive to the application level, as follows:

First, for *delta-efficient incremental checkpointing*, Kishu utilizes low-overhead live analysis (e.g., namespace patching) to track session state evolution at a novel *Co-variable* granularity (i.e., connected components of objects). Then, Kishu writes and versions Co-variables with the *checkpoint graph* representing the user workflow in terms of cell executions to minimize delta storage overhead.

Second, for *correct incremental checkout*, Kishu identifies the difference between the current and target states at the aforementioned Co-variable granularity according to the checkpoint graph. Then, it *replaces* (only) Co-variables that need updating in the state by loading data from appropriate incremental checkpoints, minimizing data load time for checkout and transparently restoring the state in the same kernel process without interruption, at *sub-second latency*.

Third, Kishu achieves generalizability and fault-tolerance through *fallback recomputation*. If a Co-variable cannot be stored in a checkpoint (e.g., it contains an unserializable object such as a hash [52]) or fails to load upon checkout, Kishu can efficiently reconstruct it upon checkout via finding the *shortest path* combining intermediate data loading and cell re-running according to the checkpoint graph.

***Difference from Existing Work (Table 1).*** Our work enables efficient time-traveling for computational notebooks through significantly different techniques vs. existing work. OS-level tools [7, 35] can incrementally checkpoint notebook states, but fail to exploit the fine-grained deltas of data science workflows (§2.2), cannot incrementally restore, and fail on remote objects (e.g. Ray [104], on-device data [124]). Existing application-level tools for saving state [55, 57, 94, 158] lack both the incremental checkpointing and restoration capabilities of our work. Works for versioning notebook cell code [22, 73, 145] help identify cells to rerun for restoration but do not directly enable it. Incremental checkpointing/PITR in DBMS [8, 111]/HPC [84, 95] and time-traveling DBMS [129, 136] focus on robust and fast logging of table/KV-store updates, while our work focuses on delta computation for complex, interdependent objects unique to notebooks, i.e., ***what to log*** (§2.2). Orthogonal works include notebook systems for speeding up data exploration using non-time-travel methods (e.g., code recommendation [89, 90]).

***Contributions.*** According to our motivations in §2, we implement Kishu (§3), a notebook system with the following contributions:

- **State Delta Detection.** We introduce our modeling of session state evolution at a novel *Co-variable* granularity, and our correct and efficient delta detection at this granularity. (§4)
- **State Versioning.** We introduce our delta-based session state versioning with the *Checkpoint Graph*, which enables efficient and fault-tolerant incremental checkpointing and checkout. (§5)
- **Time-traveling.** We show via experimental evaluation that Kishu's time-traveling is compatible with 146 classes from popular data science libraries and reduces checkpoint size and checkout time by up to 4.55× and 9.02×, respectively. (§7)

## 2 MOTIVATION

This section describes use cases for time-traveling notebooks (§2.1), characteristics of notebook workloads (§2.2), and, accordingly, our intuition for time-traveling (§2.3) and capturing state delta (§2.4).

### 2.1 Why is Time Traveling Useful?

Time-traveling computational notebooks can enable users to efficiently undo cell executions and perform path-based exploration.

***Undoing Cell Executions.*** Data operations can be irreversible (e.g., `df=df.drop_col('a')`), and users may want to return to the previous state if the operation outcome is undesirable [23, 72]. To enable interruption-free time-traveling, we can checkpoint the state delta to storage after each operation such that the session state prior to performing the operation can be returned to via loading the appropriate deltas. We empirically study this use case in §7.5.1.

***Path-based exploration.*** Data scientists often manually create branching, out-of-order cells, where each intended execution path consists of only a subset of notebook cells (e.g., cleaning steps for different models) [85, 127]. If we can efficiently persist all variations

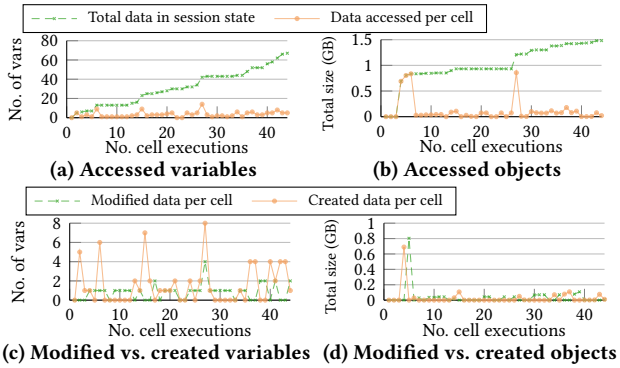


**Figure 2: Pattern of a *Sklearn* notebook [151]: (Top) many cells incrementally access a small portion of the state. (Bottom) Users balance data creation and modification.**

of objects in different paths as incremental deltas w.r.t. a shared state, users can efficiently switch paths for comparisons: only the (small amount of) data differing between paths need to be updated via loading deltas. We empirically study this use case in §7.5.2.

## 2.2 Characteristics of Notebook Workloads

Data scientists commonly use computational notebooks for exploratory work [85], which exhibit these key characteristics:

***Think Time.*** Notebook workflows often follow a sequential[2] loop of writing and running cell code, then observing output [127]. The notebook kernel can be inactive mid-loop—~10 seconds of *think time* [45] when users decide what cells to write/run next. Notebook systems use think time for computations (e.g., pre-loading data [162]). Kishu can also leverage think time for checkpointing.

***Complex Access Patterns.*** Python functions in notebooks can have complex access patterns (e.g., non-parameter variable accesses) that make them difficult and/or costly to analyze. Some works use rule-based approaches to hard-code effects of common functions (e.g., `print` not modifying the namespace) [98]. however, data scientists also often import custom functions into notebooks [127]; hence, an efficient approach that handles arbitrary functions is desirable.

***Incremental Operations.*** For fast iteration, users often run incremental cells each consisting of few lines of code and accessing few variables [85]: we observe this in on of our test notebooks, *Sklearn* (§2.2), where 40/44 cells access <10% of state data. For these workloads, Kishu's incremental checkpointing can offer larger benefits.

***Data Modifications.*** Notebook providers often limit session memory consumption [67, 83]. Hence, while creating new data, users also modify/delete existing data to conserve memory (e.g., after they have been used to generate relevant figures): in *Sklearn* (§2.2), we observe a 45:55 split between created and modified data. Kishu must efficiently undo these modifications during checkout.

## 2.3 Enabling Time Traveling

We discuss the pros and cons of incremental checkpointing and checkout approaches for enabling time-traveling to a previous state.

[2]Existing notebook systems, e.g., Jupyter, do not support concurrent cell executions.

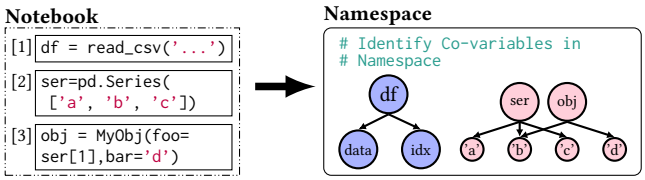


**Figure 3: Co-variables are connected components of objects. We can treat them as independent data tables.**

***OS-level Memory Snapshots.*** Tools such as CRIU [35] snapshot notebook process memory to save session state data. **(Incremental Checkpointing)** Subsequent snapshots can be made incrementally w.r.t. prior snapshots storing only dirty memory pages. However, memory-page granularity is too coarse for notebook workloads as Python data structures (e.g., lists) can be fragmented, on which operations (e.g., in-place list sorting) can cause multiple dirty pages and high checkpoint costs (§7.3). **(Complete Checkout)** Memory snapshots must be entirely loaded for state restoration and require killing the existing notebook process (to avoid PID conflicts) before restoration, which is not seamless and incurs high data loading costs (§7.5). OS-level snapshotting is also limited to single processes, hence fail on notebooks utilizing multiple/remote processors (§7.3).

***Application-level Time-Traveling (Ours).*** We use (1) application-level information to detect state deltas at a finer granularity versus memory snapshots and (2) existing data in the kernel for incremental checkout to address the drawbacks of OS-level snapshotting: **(Incremental and Generalized Checkpointing)** We track state deltas at a more logical *Co-variable granularity* (described in §2.4) by tracking in-notebook references for incremental checkpointing. For generalizability, we use objects' *reductions* as storage instructions to checkpoint multiprocessing and off-CPU workloads (§6.1). **(Incremental Checkout)** As we can directly access the notebook kernel at the application level, we can incrementally checkout by computing the difference between the current and target states (e.g., via versioning [20]) and updating only differing kernel data (§5.2).

## 2.4 Tracking State Delta for Time Traveling

We discuss pros and cons of methods of tracking the state delta at the application level for incremental checkpointing and checkout.

***Provenance-based Tracking.*** Some existing notebook systems [86, 94, 98] track state deltas via provenance-based code analysis (e.g., via ASTs [50]) at variable-level granularity. Pure static analysis requires conservativeness on identifying changed data w.r.t. control flows (e.g., `if(x<1)`) and external function calls (§2.2), causing false positives (e.g., assuming an untaken branch as taken) and large deltas; hence, these systems augment static analysis with varying levels of live instrumentation at cell runtime (e.g., resolving `x`'s value when evaluating `if(x<1)`) [98], which can result in high overhead (e.g., repeated resolutions in loops, §7.6).

***Co-variable Granularity Live Tracking (Ours).*** To avoid potential inefficiencies of provenance tracking's runtime resolutions, we propose performing *live object comparison* (i.e., comparing data pre/post-execution) only between cell executions to track per-cell

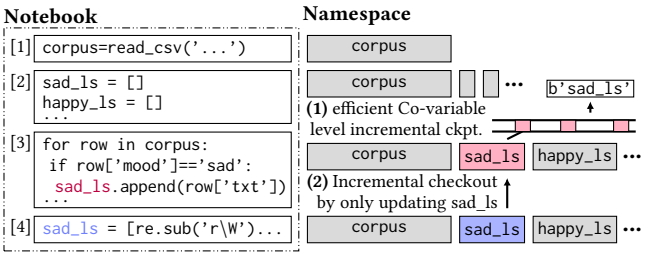


**Figure 4: Co-variable granularity deltas allows us to create size-efficient incremental checkpoints (vs. memory-page level deltas), and incrementally checkout to previous states.**

updates. Our intuition is that while doing so at variable-level granularity (like existing provenance trackers) by comparing all state objects can be costly (§7.6), it is also unnecessary as storing/loading individual variables (e.g., with variable-level KV-stores [2, 135]) for checkpoint/checkout risks breaking shared references [94]. We hence track updates at a coarser *Co-variable* level—connected components of objects (w.r.t. pointer references): we can reason from access patterns which Co-variables were possibly/surely not updated by each cell to limit object comparisons, and correctly store/load Co-variables as if they are independent data tables. Fig 3 depicts our idea: `{ser,obj}` is a Co-variable (red) as objects reachable from `ser` and `obj` overlap (`&ser[1]=&obj.foo`). `{df}` is another Co-variable (blue), and users *cannot* reach objects under `df` from objects under `ser`/`obj` via references. Notably, Co-variables are the *minimum granularity* for saving/loading state data without risking breaking shared references. §4 formally describes Co-variables and how we correctly and efficiently capture state delta at this granularity.

***Motivating Example (Fig 4).*** A data analyst performs text mining by loading the corpus (Cell 1), defining category lists (Cell 2), and sorting texts by sentiment into the lists (Cell 3). They checkpoint the state after each cell execution. **Incremental Checkpointing:** The analyst tests a mapping to clean the contained text in the lists on `sad_ls` (Cell 4, blue). Due to its interleaved construction (with other lists), `sad_ls` is fragmented; incrementally checkpointing at memory page granularity for Cell 4 (w.r.t. Cell 3) copies all pages overlapping with `sad_ls`. However, a Co-variable granularity incremental checkpoint stores only (the bytestring of) `sad_ls`. **Incremental Checkout:** The analyst undoes Cell 4's mapping due to poor results. Returning to Cell 3's state by (completely) loading a memory snapshot is slow as it also reloads the corpus. However, noting that Cell 3's and 4's states differ only by `sad_ls`, we can only load Cell 3's `sad_ls` (red) to replace Cell 4's `sad_ls` (blue) to incrementally checkout without touching the rest of the state.

## 3 SYSTEM OVERVIEW

This section presents Kishu components (§3.1) and workflow (§3.2).

### 3.1 Kishu Components

Kishu (Fig 5) interacts with notebook sessions via non-intrusive hooks, which allow Kishu to transparently (1) monitor the namespace to track session state evolution, (2) write state data to storage for checkpointing, and (3) alter the state on requested checkouts.

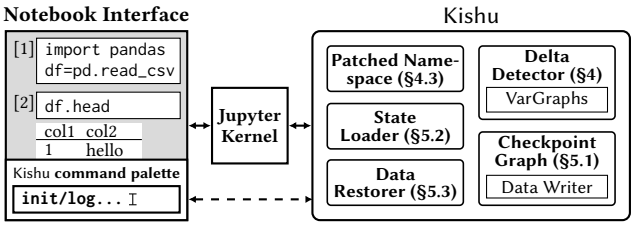


**Figure 5: Kishu architecture. It utilizes a hook to observe session state deltas and transparently write/replace data in the kernel namespace for incremental checkpoint/checkout.**

***Patched Namespace.*** On session start, Kishu *patches* the namespace to monitor accesses to its contents during cell executions (§4.3). It tracks user-referenced variable names to identify candidate Co-variables to check for updates in, which are passed to the Delta Detector to compute the Co-variable granularity state delta.

***Delta Detector.*** The Delta Detector computes the state delta based on the candidates identified from the Patched Namespace (i.e., which of the candidate Co-variables were actually updated by the cell execution). We discuss the Kishu's delta detection in (§4).

***Checkpoint Graph.*** The Checkpoint Graph is a tree-like structure analogous to Git's commit graph [63], in which Kishu writes, stores, and versions incremental checkpoints consisting of the updated Co-variables (i.e., the state delta) of each cell execution (§5.1). The incremental checkpoints stored in the Checkpoint Graph are used by the State Loader to perform incremental checkout.

***State Loader.*** The State Loader restores to a session state upon requested checkout. It first identifies the difference between the current (i.e., existing items in the namespace) and target states via the Checkpoint Graph, then loads necessary data from the Checkpoint Graph to replace Co-variables that need updating (§5.2).

***Data Restorer.*** The Data Restorer is a mechanism that utilizes fallback recomputation to restore missing data for checkout (e.g., Kishu failed to serialize the data during prior checkpointing). It reconstructs missing data by combining loading dependent data and cell re-runs according to the Checkpoint Graph. (§5.3)

### 3.2 Kishu Workflow

This section covers Kishu's operations during a notebook workflow. Users interact with Kishu with the in-Jupyter Command Palette (Fig 5), such as to *attach* it to a new notebook session;[3] Kishu then monitors the namespace for state deltas, checkpoint after each cell execution, and perform on-demand checkout to previous states.

***Attaching Kishu to a Notebook Session.*** When initializing a notebook session, **`init`** attaches Kishu to the kernel, which will patch the namespace and initialize the Checkpoint Graph on storage.

***Incremental Checkpointing.*** After each cell execution, the Delta Detector uses the Patched Namespace to identify updated Co-variables and stores them in a new incremental checkpoint/node with a unique `ckpt_id` on the Checkpoint Graph. The user may view the stored graph, checkpoints, and their IDs with **`log`**.

[3] A depiction of Kishu's interface can be found in our prior work's demo paper [92].

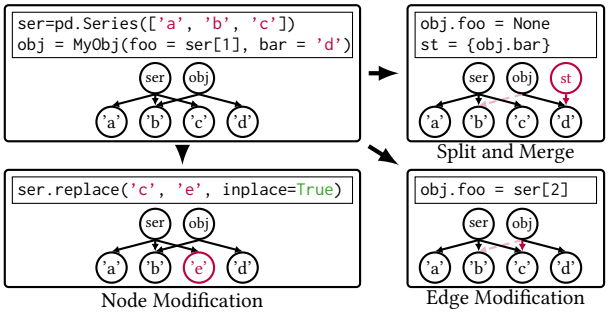


**Figure 6: Three ways which the Co-variable {ser,obj} (first appearing in Fig 3) can be updated by a cell execution (red).**

***Incremental Checkout.*** Kishu will restore a previous session state with **checkout** `ckpt_id`. The State Restorer identifies differing Co-variables between the current and target states according to the Checkpoint Graph, then accordingly loads only the necessary data for restoration. If necessary, the Data Restorer reconstructs data that is missing or failed to load via fallback recomputation.

# 4 ACCURATE AND FAST DELTA DETECTION

This section describes how Kishu correctly detects Co-variable granularity state deltas necessary for incremental checkpointing and checkout (§2.4). We formally describe Co-variables in §4.1 and how we detect Co-variable updates correctly (§4.2) efficiently (§4.3).

## 4.1 Co-variables

This section introduces Kishu's definition of the Co-variable.

***Preliminary.*** In Python and the Jupyter Notebook ecosystem, variables and objects are 2 distinct concepts: A **variable** is a named entity from which various **objects** are *reachable*—for an example list `ls=[1,2,3]`, the list name (`ls`) is a variable and each list element (1, 2, 3) is an object. We define reachability reference-wise, i.e., object `y` is reachable from variable `x` if `y` can be accessed from `x` through a chain of references. Some common reachability patterns include *subscripting* (e.g., `y=x[0]`) and *class member* (e.g., `y=x.attr`). Given this distinction between variables and objects and reachability definition, we now define Co-variables as follows:

**Definition 1.** A **Co-variable** is a set of variable names $\mathcal{X} = \{x_1, ..., x_i\}$ from which the reachable objects form a **maximally connected component**. That is, for any variable $y$ not in the set, the objects reachable from $x_1, ..., x_i$ are not reachable from $y$.

A Co-variable can consist of one name (e.g., a primitive, `x=1`) or multiple names from which same objects can be reached (i.e., shared references). For example, in Fig 6, the string object `'b'` is reachable from both Pandas Series `ser` and object `obj` via subscript and class member respectively, hence `{ser,obj}` is a Co-variable. Co-variables are self-contained by definition, i.e., there are no inter-Co-variable references. They can be modified by cell executions:

**Definition 2.** A Co-variable $\mathcal{X} = \{x_1, ..., x_i\}$ is **modified** by a cell execution if the graph structure of the connected component of objects reachable from $x_1, ..., x_i$ is modified, counting both node (i.e., object) and edge (i.e., reference) additions and deletions.

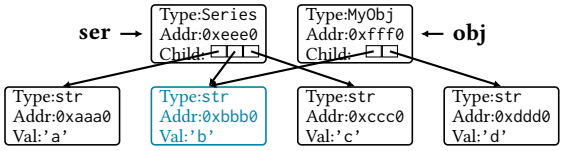


**Figure 7: VarGraphs of `ser` and `obj` intersect from shared reference to `'b'` (blue), hence `{ser,obj}` is a Co-variable.**

For example, the Co-variable `{ser,obj}` in Fig 6 is modified node-wise with an in-place update "`ser.replace`" **(bottom)**, and edge-wise with a re-assignment "`obj.foo=ser[2]`" **(bottom-right)**. Co-variables can be created and deleted via *split and merge* **(right)**: `{obj,ser}` is deleted via a split as `obj` and `ser` longer share references, and `{obj,st}` is created via a merge. We collectively refer to Co-variable modifications, creations, and deletions as *updates*; the Co-variables updated by a cell execution form its *state delta*.

## 4.2 Accurate State Delta Detection

This section describes how Kishu accurately detects Co-variable membership (i.e., which variables form a Co-variable) and updates.

***VarGraphs.*** Kishu detects Co-variable membership and updates with VarGraphs—a graph structure constructed on each variable that captures its reachable objects in the namespace. Fig 7 shows an example: each node in a variable's VarGraph corresponds to a reachable object, containing the (1) type, (2) memory address, and one of (3) pointers to other reachable objects (i.e., children) for non-primitives, or (4) value for primitives. For example, the node for `list` contains 3 child pointers to the 3 nodes for strings `'a'`, `'b'`, and `'c'`, and the node for string `'b'` holds its value `'b'`. [4]

***Detecting Co-variable membership.*** Co-variable membership is determined by intersecting VarGraphs. For example, in Fig 7, `ser` and `obj` form a Co-variable as the node `'b'` is in both graphs (red).

***Detecting Co-variable updates.*** Co-variable updates is determined by comparing VarGraphs before and after cell executions. A graph structure modification and/or a node attribute change (e.g., object memory address or type) indicates an update to the Co-variable.

***Accuracy Guarantee.*** As Kishu constructs VarGraphs following object rechability, it detects Co-variable updates with no false negatives (empirically verified in §7.2.1). However, Kishu's update detection is *conservative*: there may be *false positives* if objects are dynamically generated (e.g., datatype objects) with a different memory address each time during VarGraph construction/object traversal, or cannot be traversed into (i.e., lacking referencing instructions, e.g., generators [51], which Kishu assumes to be updated on access).

## 4.3 Efficient State Delta Detection

This section describes how Kishu speeds up Co-variable update detection. Identifying updates across the entire global namespace via VarGraphs can be expensive (due to object traversals); Kishu needs to reduce the number of Co-variables (i.e., portion of namespace) it checks after cell executions without reducing detection accuracy.

[4] The VarGraph is inspired by ElasticNotebook's *ID graph* [94] which captures reachable objects' memory addresses; VarGraphs uniquely contain datatypes and primitive values for additional robustness (e.g., detecting a different primitive in the same address).

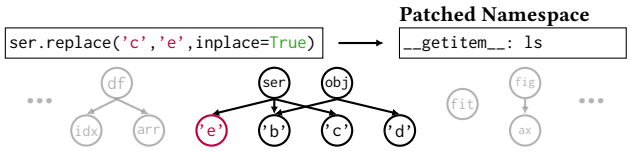


**Figure 8: Kishu efficiently captures state delta via Patched Namespace: it only needs to check Co-variable `{ser,obj}` for updates, as other Co-variables surely weren't updated.**

***Identifying Possibly Updated Co-variables.*** Cell executions in Jupyter Notebook interact with the global namespace (i.e., `globals()`). Therefore, if Kishu can capture variable references in the cell execution, it can reason about which Co-variables were possibly updated (and which ones were definitely not), as follows:

**Definition 3.** A Co-variable $\mathcal{X} = \{x_1, ..., x_i\}$ is **accessed** by a cell execution if any variable $x_1, ..., x_i$ is accessed (via getting, setting, or deletion) during the cell execution.

Co-variable accesses indicates *possible* updates (e.g., via a *member function call* `ser.replace`). Kishu patches the accessor, setter, and deletion methods of the global namespace (Fig 8) to capture variable (hence Co-variable) accesses, which helps identify the *possibly updated Co-variables*: if the members of a Co-variable $\mathcal{X}$ overlaps with the cell execution's accessed variables, then it may have been updated (e.g., `{ser,obj}` in Fig 8): Kishu will verify the update by (1) re-generating VarGraphs for its member variables, (2) comparing the VarGraphs with those before the cell execution to identify modifications, and (3) intersecting the VarGraphs amongst variables of accessed Co-variables to identify merges and splits. Otherwise, the Co-variable surely wasn't updated and Kishu skips its check for this cell execution (e.g., `{df}` in Fig 8, greyed out).

**Lemma 1.** A Co-variable $\mathcal{X} = \{x_1, ..., x_i\}$ can be updated by a cell execution only if at least one of $x_1, ..., x_i$ was accessed in the code.

Proof. Suppose not, i.e., Co-variable $\mathcal{X}$ does not intersect the accessed variables and was updated. Then, $\mathcal{X}$ must have been updated through via variable $y$ that was not part of $\mathcal{X}$ before the start of the cell execution. Due to Co-variables' self-containment (§4.1), objects reachable from $\mathcal{X}$ cannot possibly be accesed via $y$ during the cell execution without creating a reference by using one of $x_1, ..., x_i$ first (e.g., $y.foo = x\_i$), but doing so violates our assumption. □

As only a small portion of variables are accessed per cell in a typical data science notebook, Kishu significantly reduces delta detection overhead with this approach (empirically verified in §7.6).

***Remark.*** As Kishu patches the notebook session's global namespace, it is impossible for users to use variables from within the notebook (e.g., to modify objects) undetected. Hence, Kishu will not misidentify Co-variables possibly updated via references. Users may still use non-referencing methods to update data such as C-pointer-based modifications, but these cases are rare in notebooks (found in 0/60 surveyed notebooks [94]) Some libraries do perform memory-based updates (e.g. NumPy's slicing [142]). However, the objects are supported by Kishu as these updates are still invoked via referencing (e.g., `arr[0,1] += 1`, empirically verified in §7.2.1).

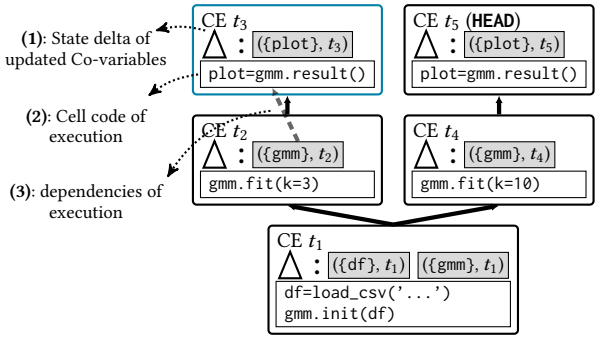


**Figure 9: A Checkpoint Graph with 2 branches ($t_1 \rightarrow t_2 \rightarrow t_3$ and $t_1 \rightarrow t_4 \rightarrow t_5$). Kishu manages state deltas in the Checkpoint Graph; Co-variables are versioned by update time.**

## 5 INC. CHECKPOINT & CHECKOUT

This section describes Kishu's efficient time-traveling with the Co-variable granularity state deltas. We describe Kishu' incremental checkpointing in §5.1, Kishu's incremental checkout in §5.2, and how Kishu time-travels to and from notebook states with problematic (e.g., unserializable) data in a fault-tolerant manner in §5.3.

### 5.1 Incremental Checkpointing

This section describes how Kishu performs incremental checkpointing by writing and managing per-cell-execution checkpoints containing the updated Co-variables with the *Checkpoint Graph*.

***Checkpoint Graph.*** The Checkpoint Graph is a directed tree of (incremental) checkpoints representing the branch-based state evolution. Nodes are added for each of Kishu's checkpoints, and are timestamped with the completion time $t$ of the corresponding cell execution (we refer to the timestamped node and cell execution as *node* $t$ and *CE* $t$, respectively). The Checkpoint Graph maintains a *head node* tracking the current state. Each node $t$ contains the state delta consisting of Co-variables updated by CE $t$. Co-variables stored in each node $t$ are versioned accordingly:

**Definition 4.** A **Versioned Co-variable** is a Co-variable-timestamp pair $(\mathcal{X}, t)$ representing the Co-variable $\mathcal{X}$ updated by CE $t$.

Versioned Co-variables are analogous to versioned datasets: the same Co-variable (w.r.t. variable membership) can take on multiple values during a session being updated by different cell executions. Fig 9 show an example: CE $t_3$ creates the Co-variable `{plot}`, which is stored in node $t_3$ (red) as the *Versioned Co-variable* $((plot), t_3)$.

***Writing into the Checkpoint Graph.*** After each CE $t$, Kishu writes a new node $t$ in the Checkpoint Graph with (1) the versioned Co-variables in CE $t$'s state delta, (2) CE $t$'s code, and (3) CE $t$'s accessed versioned Co-variables stored in previous checkpoints (§4.3). For example, node $t_3$ in Fig 9 (blue) contains the code ("`plot=gmm.result()`") and its dependency on $((gmm), t_2)$ from node $t_2$ (dashed line). Notably, the state delta, code, and variable accesses are respectively analogous to the *update*, *operation*, and *dependencies* in database logging and versioning. The new node $t$ is written under the head node $s$, and and a parent-child relation is added from $s$ to $t$ (now the new head node).

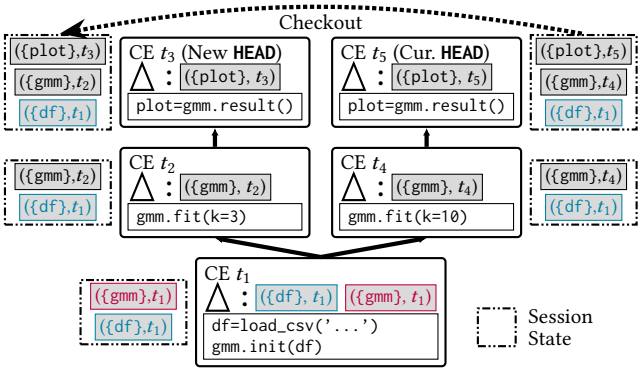


**Figure 10: Checkout from session state $t_5$ to session state $t_3$: The versions of `df` (blue) is identical between the branches. The versions of `gmm` (red) has diverged and needs updating.**

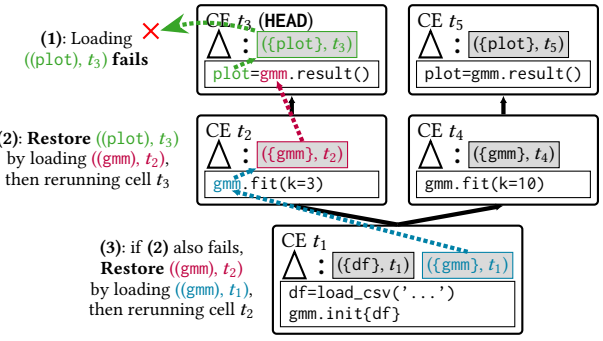


**Figure 11: Fallback recomputation for ({plot}, $t_3$) (green). It can be recomputed by loading ({gmm}, $t_2$), then rerunning cell $t_3$ (red). If ({gmm}, $t_2$) also fails to load, it can be recomputed by loading ({gmm}, $t_1$) and rerunning cell $t_2$ (blue).**

***Handling Unserializable Data.*** If Kishu cannot write an updated Co-variable into the Checkpoint Graph (e.g., it contains an unserializable object such as a generator [51] or hash [52]), Kishu simply skips its storage. Instead, upon checkout, the missing Co-variable will be restored through fallback recomputation enabled by the cell code and dependencies stored in the Checkpoint Graph node (§5.3).

## 5.2 Efficient State Restoration

Kishu's goal for incremental checkout is to accurately and efficiently restore the current state to the target state. To do so, it must identify the contents of the target state via its timestamp, analogous MVCC's timestamped snapshots [20]; instead of versioned tables, we identify *Versioned Co-variables* in the (timestamped) target state:

**Definition 5.** The **Session State** at timestamp $t$ is a set of $n$ Versioned Co-variables $\{(\mathcal{X}_i, t_i) | 1 \leq i \leq n\}$ such that for each $(\mathcal{X}_i, t_i)$:

1. $t_i$ is an ancestor of $t$ on the Checkpoint Graph.
2. There must not exist another versioned Co-variable $(\mathcal{Y}_j, t_j)$ such that $\mathcal{X}_i \cup \mathcal{Y}_j \neq \emptyset$ and $t_j$ is a child of $t_i$ and ancestor of $t$.

The session state at timestamp $t$[5] (*state* $t$ for brevity) is the set of all Versioned Co-variables that are in the namespace after CE $t$, i.e., not overwritten by a newer Versioned Co-variable prior to CE $t$. For example, in Fig 10, state $t_3$ (top-left) consists of $(\{plot\}, t_3)$, $(\{gmm\}, t_2)$, and $(\{df\}, t_1)$. It does not contain $(\{gmm\}, t_1)$ as it was overwritten by CE $t_2$ (`gmm.fit(k=3)`) which writes $(\{gmm\}, t_2)$. Each state $t$ dictates which Versioned Co-variables should be loaded from various Checkpoint Graph nodes for checkouts; for efficient incremental checkout, Kishu identifies the current and target states' *difference* w.r.t. the (versioned) Co-variables that need updating: some Co-variables do not need updating when converting the current state to the target state, identifiable via the Checkpoint Graph:

**Definition 6.** A Co-variable $\mathcal{X}$ is **identical** between the current state $t_a$ and target state $t_b$ if a Versioned Co-variable $(\mathcal{X}, t_c)$ exists in the session states of $t_a$, $t_b$, and $t_c$, where $t_c$ is the **lowest common ancestor** of node $t_a$ and node $t_b$. Otherwise, if no such $(\mathcal{X}, t_c)$ exists, then the Co-variable $\mathcal{X}$ has **diverged** between $t_a$ and $t_b$.

[5]Kishu stores snapshots of Session State metadata (i.e., references to contained Co-variables) in Checkpoint Graph nodes.

A Co-variable $\mathcal{X}$ is identical between states $t_a$ and $t_b$ if its versioned counterpart is consistent across $t_a$, $t_b$, and $t_c$, i.e., no CE between (1) nodes $t_a$ and $t_c$ and (2) nodes $t_b$ and $t_c$ updated $\mathcal{X}$, hence does not need updating when checking out from $t_a$ to $t_b$. For example, in Fig 10, if checking out from $t_5$ to $t_3$, the Co-variable `{df}` (blue) is identical between the states as no CE between (1) nodes $t_1$ and $t_3$ (2) nodes $t_1$ and $t_5$ updated it. Otherwise, if the Co-variable $\mathcal{X}$ has diverged between the current and target states, it will need updating (by either loading an appropriate Versioned Co-variable or deleting it) to checkout to the target state. For example, the Co-variable `{gmm}` (red) has diverged between nodes $t_5$ and $t_3$ as their parents ($t_4$ and $t_2$) both updated `gmm` with their CE (fitting with `k=3` and `k=10`), hence, `gmm` (and `plot`) needs updating via loading `({gmm},`$t_2$`)` if checking out from state $t_5$ to state $t_3$.

***Performing State Checkout.*** When checking out to the state at node $t$, The State Restorer (§3.1) performs the following steps:

1. Load the appropriate Versioned Co-variables from nodes (i.e., node $t$ and ancestors of node $t$) to update diverged Co-variables between the state of the current head node $s$ and node $t$.
2. Update/re-generate VarGraphs (§4.2) for updated Co-variables.
3. Move the head from node $s$ to the checked out node $t$.

Notably, the next cell execution will create a node in a new branch rooted at $t$ in the Checkpoint Graph, e.g., the graph in Fig 10 is generated through the sequence $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow$ (checkout to $t_1$) $\rightarrow t_4 \rightarrow t_5$. If during checkout, a required Versioned Co-variable is missing (i.e., due to serialization failure, §5.1) or fails to load (i.e., deserialization failure), Kishu restores it via fallback recomputation.

## 5.3 Robust Restoration

In this section, we describe how Kishu restores problematic data to achieve generalizable and fault-tolerant incremental checkout.

***Fallback Recomputation.*** As each Checkpoint Graph node $t$ contains the code of CE $t$ and (2) which previous Versioned Co-variables $(\mathcal{X}_j, t_j)$ CE $t$ accessed (§5.1), any Versioned Co-variable in node $t$'s state delta can be recomputed by (1) loading accessed Versioned Co-variables from previous checkpoints, then re-running CE $t$. For example, in Fig 11, suppose ({plot}, $t_3$) (green) fails to load when checking out to state $t_3$. ({gmm}, $t_2$) is required to rerun CE $t_3$ (red);

therefore, ({gmm}, $t_2$) is loaded from the parent node $t_2$, and after rerunning CE $t_3$ on the input ({gmm}, $t_2$), ({plot}, $t_3$) is restored.

***Dynamic and Recursive Fallbacks.*** Kishu's fallback recomputation is dynamic and recursive—if another Co-variable is missing or fails to load when retrieving recomputation inputs, fallback recomputation can be recursively performed for that Co-variable. For example, if ({gmm}, $t_2$) from node $t_2$ fails to load (as fallback recomputation for ({plot}, $t_3$) from node $t_3$), it itself can be recomputed by loading ({gmm}, $t_1$) from node $t_1$ and rerunning CE $t_2$ (blue).

***Remark.*** Kishu guarantees *exact* restoration for all serializable Co-variables w.r.t. *bytestring representation* if there are no hidden serialization errors (§6.2). While Kishu's fallback recomputation restores some problematic Co-variables, exactly restoring Co-variables that both (1) fail to store/load and (2) are created non-deterministically (e.g., random generators) is currently unsupported. This limitation is similarly seen in Spark [165] and Ray's [104] lineage-based fault tolerance; however, in our case, such objects are rare in data science libraries (§7.2), hence we consider this limitation to be acceptable.

# 6 IMPLEMENTATION AND DISCUSSION

This section describes Kishu's implementation details (§6.1) and design considerations (§6.2).

## 6.1 Implementation

***Integrating with Jupyter.*** Kishu is implemented as a separate application from the notebook process, usable without altering the base Jupyter application. Upon session initialization, Kishu places hooks into the kernel (`pre_run_cell` and `post_run_cell` [78]) and patches the namespace (`user_ns` [77]) (§3.1), allowing the standalone Kishu process to detect state deltas, write data to storage, and overwrite data in the namespace upon checkout transparently.

***Serialization Protocol.*** The Pickle protocol (i.e., `__reduce__` [55]) is used for (1) object serialization and (2) constructing VarGraphs for identifying Co-variables, i.e., object `y` is reachable from another object `x` if `pickle(x)` includes `y`. As Pickle is the de-facto standard (in Python) observed by almost all libraries (e.g., NumPy, PyTorch [47]), Kishu can be used in almost all cases. Kishu's per-Co-variable storage also enables *mixing and matching* serialization libraries for coverage: Currently, Kishu will try CloudPickle [32] first, then use Dill [57] as a fallback for Co-variables that CloudPickle fails on.

***Storing Checkpoints.*** Kishu uses SQLite [139] to store Versioned Co-variables in the Checkpoint Graph. However, any storage mechanism [29–31] can be used in its place—even in-memory ones if the user wants to maximize checkpointing/checkout efficiency.

## 6.2 Design Considerations

***Silent Serialization Errors.*** Certain object classes may contain incorrect serialization instructions, which, despite being able to be stored/loaded to/from storage, result in silent errors. Kishu currently assumes that instructions are correctly implemented for all objects w.r.t. equality before and after pickling, and does not prevent these silent errors. However, these cases are rare (§7.2.1), and Kishu provides a blocklist file for users to force fallback recomputation for Co-variables containing objects belonging to these classes.

**Table 2: Summary of Notebooks for Evaluation.**

| Notebook | Topic | Library | Cells | Time(s) | Data(MB) | Final |
|---|---|---|---|---|---|---|
| Cluster[40] | Cluster analysis | seaborn[157] | 24 | 1703 | 43 | Yes |
| TPS[154] | Random forest | intelex[132] | 49 | 154 | 31 | Yes |
| Sklearn[151] | Text mining | sklearn[130] | 44 | 512 | 185 | No |
| HW-LM[62] | Linear regression | NumPy[142] | 81 | 13 | 1 | Yes |
| StoreSales[18] | TS analysis | SM[141] | 41 | 665 | 122 | Yes |
| Qiskit[15] | Quant. Computing | Qiskit[125] | 85 | 46 | 1 | No |
| TorchGPU[88] | Image classification | PyTorch[56] | 27 | 716 | 1090 | Yes |
| Ray[126] | Distrib. Computing | Ray[9] | 20 | 2361 | 92 | No |

**Table 3: Categorization of 146 Object Classes for Evaluation. Referred to in Fig 12, Table 4, and Table 5.**

| Category | Example Libraries | Example Class |
|---|---|---|
| Data Analysis | pandas[109], polars[119], pyarrow[11] | pd.DataFrame[110] |
| Data Visualization | matplotlib[147], plotly[118], seaborn[157] | plt.Figure[42] |
| Machine Learning | sklearn[130], xgboost[161], scipy[133] | GMM[131] |
| Deep Learning | tensorflow[148], torch[56], keras[66] | torch.Tensor[124] |
| NLP | nltk[122], textblob[97], wordcloud[106] | TextBlob[97] |
| Computer Vision | photutils[41], torchvision[34] | ImageDepth[116] |
| Dist. Computing | pyspark[137], ray[9], optuna[33] | pyspark.sql[138] |
| Data Pipelining | huggingface[48], transformers[150] | BertTokenizer[75] |

***Alternative Delta Detection Methods.*** While Kishu's VarGraphs can generalizably detect Co-Variable updates, there are specific cases that allow for more efficient detection methods such as (1) hashing (e.g., XXH64 [164]) for array-likes [142]) and (2) rule-based static cell (e.g., `df.head`) identification for skipping update detection (§2.2). Kishu currently uses hashing for common array-likes (e.g., tensors [148]), but can be extended to incorporate (1) other detection methods for specific classes and (2) rule-based cell analyses.

# 7 EXPERIMENTAL EVALUATION

In this section, we empirically study the effectiveness of Kishu's time-traveling. We make the following claims:

1. **Generalizable and Robust Mechanism:** Kishu can identify modifications to, and correctly restore session states containing 146 object classes from common Data Science libraries. (§7.2)
2. **Low Checkpoint Storage Cost:** Kishu's optimizations result in its per-cell-execution checkpoints being up to 4.55× smaller compared to those from the next best mechanism. (§7.3)
3. **Low Checkpoint Times:** Kishu's checkpoints are created up to 5.12× faster compared to the next best mechanism. (§7.4)
4. **Fast Incremental Checkout:** Kishu's novel incremental restoration is crucial to its *sub-second* checkout times — up to 8.18× and 4.18× faster than the next best mechanism for undoing cell executions and switching branches, respectively. (§7.5)
5. **Low Overhead Delta Detection:** Kishu incurs negligible runtime overheads on data science notebooks for capturing the state delta — less than 3.0% of the notebook session runtime and up to 4.08× less than alternative tracking approaches. (§7.6)

## 7.1 Experiment Setup

***Datasets.*** We select 8 data science notebooks from Kaggle Grandmaster-level users or Github-hosted tool tutorials (e.g., Ray) (Table 2), each featuring a popular data science library, which we categorize as *in-progress* (3/8) or *final* (5/8), with the former containing out-of-order


**Figure 12: Checkpoint/checkout failures. Kishu successfully checkpoints/checkouts all object classes with no failures.**

cell executions and the latter lacking them.[6] Notably, the final notebooks are also cleaned and contain memory/runtime optimizations. We empirically verify all notebooks follow traits discussed in §2.2.

We also select 146 common data science library classes ( Table 3), on which we evaluate Kishu's correctness and robustness.

***Methods.*** We evaluate Kishu against existing tools capable of enabling time-travelling on notebooks to various degrees:

- CRIU [35]: Performs a system-level memory dump of the process hosting the notebook session. The session state is restored by loading the memory dump and reviving the process.
- CRIU-Incremental [35]: CRIU with snapshot deduplication, storing only dirty memory pages in subsequent snapshots.
- DumpSession [58]: An application-level checkpointing tool that serializes the entire session state into one single file.
- ElasticNotebook [94]: An application-level notebook migration tool that balances data serialization and cell recomputation to achieve optimized session replication times.
- Kishu+Det-replay: A checkpoint-optimized version of Kishu based on operation replay [101, 108] that uses manual annotation[7] to skip incremental checkpointing for deterministic cells. These deterministic cells are replayed as necessary on checkout.

***Ablation Study.*** We additionally compare the overhead of Kishu's update detection mechanism with these tracking methods (§7.6):

- IPyFlow [134]: A hybrid dynamic-static (i.e., AST analysis with live symbol resolution) for obtaining sub-variable (i.e., symbols, e.g., `ls[x]`) level granularity to perform reactive cell executions.
- AblatedKishu (Check all): Always perform update detection for all Co-variables in the session state after each cell execution, regardless of whether they were accessed in the previous cell.

***Methodology and Measurement.*** We run notebook cells sequentially from top to bottom and checkpoint after each cell execution. We checkout into the same state for Kishu and Kishu+Det-replay and into a fresh kernel/process for other methods. We measure the (1) *checkpoint time* (including both tracking and data writing) after each cell execution, (2) *checkout time* to restore the state from checkpoint files, and (3) *tracking overhead* of Kishu after each cell execution to track updates. We clear the page cache between runs.

***Environment and Reproducibility.*** Experiments are performed on an Ubuntu server with 2 AMD EPYC 7552 48-Core Processors and 1TB RAM. All checkpoints are written to a mounted NFS, with disk read and write speeds of 519.8 MB/s and 358.9 MB/s, respectively. Our Github repository[8] contains our implementation of Kishu, experiment notebooks/library classes, and scripts.

[6]We discuss this categorization and its implications in detail in our technical report [93].
[7]Automatically detecting non-/determinism in executions is out of scope for this work.
[8]https://github.com/illinoisdata/kishu-vldb

**Table 4: Kishu handles these classes existing works fail on.**

| Tool | Description | Failure Classes |
|---|---|---|
| CRIU | Dist. Computing | pyspark.sql [138], ray.Dataset [143] |
| | On-device data | tf.tensor [149], torch.tensor [124] |
| | Data Pipelining | Pipeline [76], BertTokenizer [75] |
| DumpSession | Unserializable Data | pl.LazyFrame [120], bokeh.figure [21] |

**Table 5: Summary of Kishu's update detection.**

| Result | Description | Count |
|---|---|---|
| Success | Kishu reports an update when object is changed | 120 |
| False Positive | Kishu reports update on access when object is unchanged | 14 |
| Pickle Error | Object can't be deterministically stored, Kishu reports update | 12 |
| Fail | Object is changed but Kishu does not report an update | 0 |

## 7.2 Generalized and Robust Time Traveling

This section compares the robustness of Kishu's time-traveling to existing methods. We attempt to checkpoint and checkout session states containing objects from the 146 data science library classes and compare number of classes each method fails to checkout.

We report results in Fig 12. Kishu completes time-traveling for all 146 libraries, handling 6 classes with multiprocessing and/or off-CPU data and 7 unserializable classes that CRIU and DumpSession fail on, respectively: unlike CRIU, Kishu utilizes reductions (§6.1) to store Co-variables, hence it can store distributed or off-CPU data (e.g., Ray's dataset[143] or on-GPU tensors[124, 149]) and unlike DumpSession, Kishu's fallback recomputation allows it to restore Co-variables with (1) unserializable objects (e.g., pl.LazyFrame[120]) or (2) serializable objects that can't deserialize (e.g., bokeh.figure[21]). Table 4 summarizes these noteworthy classes.

*7.2.1 Accurate Delta Detection.* We verify Kishu's delta detection accuracy by comparing two VarGraphs generated for each class object before and after (1) updating a class attribute (e.g., `model.key = 'A'`) or (2) updating nothing. We count the number of VarGraph differences for case (1) as *successes* and case (2) as *false positives*.

We report results in Table 5. Kishu's VarGraphs accurately captures object updates in 120 classes. While Kishu reports false positives in 14 classes, (e.g., due to dynamically generated reachable objects), they only affect Kishu's efficiency (i.e., during time-traveling); however, Kishu maintains accuracy by considering these objects to be updated on access. We also find that 12 classes contain *silent pickling errors* (§6.2); nevertheless, Kishu reports these objects to be updated on access similar to false positives, and users may force their (fallback) recomputation if needed (§6.2). Notably, Kishu has no false negatives: Kishu will always report if an object is changed.

## 7.3 Small Incremental Checkpoint Sizes

This section compares Kishu's checkpoint sizes with those of existing tools: we checkpoint the session state after each cell execution with each method and measure the total storage size of checkpoints.

We report results in Fig 14. Kishu's cumulative checkpoint size is consistently the smallest (expectedly except for Kishu+Det-replay, explained shortly) and is up to 4.55× smaller than the next best alternative (*HW-LM*). ElasticNotebook, while the next best method

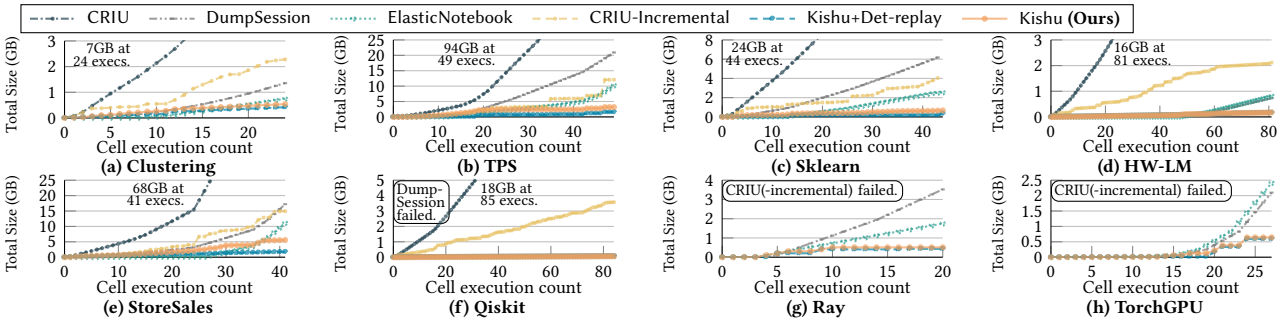


**Figure 13: Kishu's cumulative incremental checkpoint storage costs compared to checkpoint storage costs of existing tools. Kishu's incremental checkpoints are consistently the smallest, and is up to 4.55× smaller than the next best alternative.**

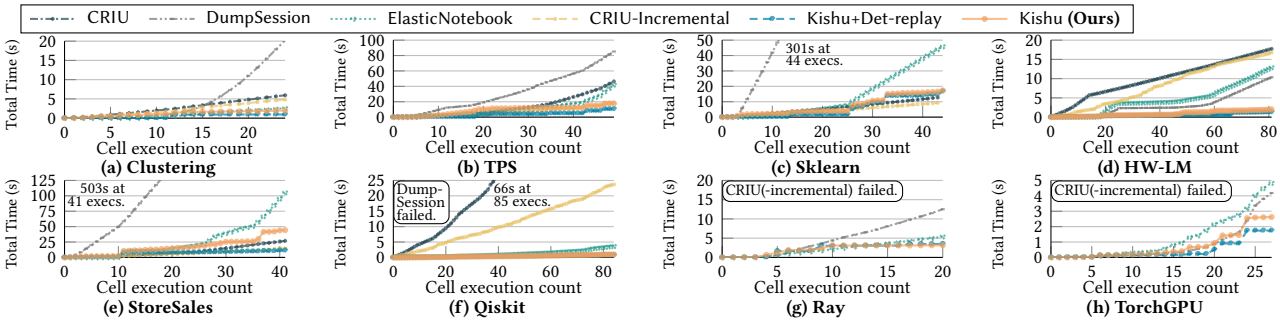


**Figure 14: Kishu's cumulative incremental checkpoint time compared to existing tools. Kishu's incremental checkpointing incurs overhead of only up to 15.5% of notebook runtime and can be up to 5.12× faster than the next best alternative.**

on 6/8 notebooks and also has fault-tolerant mechanisms to checkpoint all 8 notebooks, can fall short in checkpointing time (§4.3). CRIU-Incremental, while also incrementally checkpointing, is not the next best method on any notebook, losing to ElasticNotebook and DumpSession on 6 and failing to checkpoint on 2 as it (1) incrementally checkpoints at the coarser memory page level (§2.4), and (2) does not handle off-CPU data and multiprocessing (§7.2). DumpSession fails on *Qiskit* as it cannot handle unserializable data, and CRIU incurs prohibitive storage costs (94GB, *TPS*) as it non-incrementally checkpoints at OS-level. While Kishu+Det-replay can save checkpoint storage cost of up to 3.95× vs. Kishu (*StoreSales*) by skipping checkpointing after deterministic cells, it (1) needs manual annotation and (2) can result in unacceptable checkout times (§7.5).

## 7.4 Low Incremental Checkpoint Time

This section compares the checkpoint time of Kishu with that of existing tools: we measure the total time spent by each method creating checkpoints after each cell execution.

We report results in Fig 13. Kishu's cumulative checkpointing time is the lowest (except for Kishu+Det-replay) on 5/8 notebooks, being only up to 15.5% of notebook runtime (*HW-LM*) and up to 5.12× faster (*HW-LM*) than the next best alternative on these notebooks. While CRIU-Incremental checkpoints faster than Kishu on 3/8 notebooks owing to memory dumping being faster than serialization for unit data, the difference is negligible (up to 3.03×, *StoreSales*) compared to the reliability issues (§7.2), space inefficiency (§7.3), and slow checkout times (§7.5). Compared to ElasticNotebook, Kishu's checkpointing is EAFP-based [65]: if it fails to store a Co-variable, it will simply recompute it upon checkout via fallback recomputation. This allows it to skip the profiling (i.e., for data sizes and serializability) required for ElasticNotebook's optimization (for what to store/recompute), which causes checkpoint times slower than DumpSession on 2/8 notebooks.

## 7.5 Fast Incremental Checkout

This section compares the efficiency of Kishu's incremental checkout with the (non-incremental) checkout of existing methods. We generate per-cell-execution checkpoints on the notebooks following the methodology in §7.3 and §7.4, then measure the time it takes for each method to checkout to a previous state (i.e. undo, §7.5.1) or checkout to a different execution branch (§7.5.2).

*7.5.1 Fast Execution Undo.* For each notebook, we measure the time it takes to undo various dataframe and plot operation cells.

We report the results in Fig 15a. Kishu is the only method capable of incrementally checking out: it achieves *sub-second* cell execution rollbacks on all test cases, and is up to 8.18× faster than the next best alternative (*StoreSales*). While CRIU-Incremental achieves checkpoint times comparable with Kishu, it is up to **36**× slower for checking out (*StoreSales*) and the slowest method for undos on 5/6 notebooks, due to it needing to piece together the memory snapshot of the notebook process to restore from multiple (incremental) checkpoint files. CRIU, DumpSession, and ElasticNotebook cannot

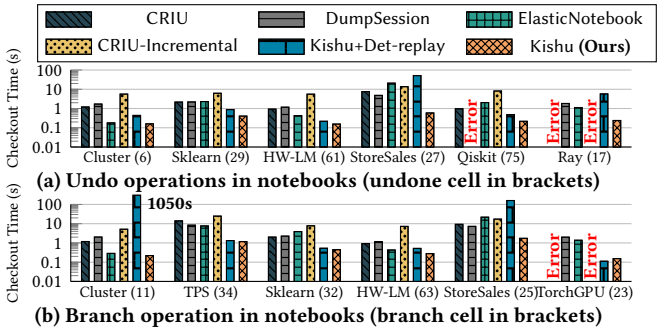


**Figure 15: Methods' checkout time for undoing executions (top) and switching to a branched states (bottom). Kishu's checkout is up to 8.18× and 4.18× faster, respectively, than the next best alternative; notably, the former is *sub-second*.**

incrementally checkout hence they cannot consistently perform sub-second undos. For example, the *Sklearn* notebook test case drops a column in an *auxiliary dataframe* that is 1.4MB in size (vs. the 133MB main dataframe). Kishu identifies that it only needs to load the auxiliary dataframe from before the cell execution and undoes the operation in 0.4 seconds; however, other non-Kishu+Det-replay methods all require the entire session state to be overwritten with a complete load of checkpoint data, taking an upwards of 6 seconds to do so (and in CRIU and CRIU-Incremental's case, also killing and restarting the current notebook process).

*7.5.2 Fast Path Exploration.* For each notebook, we (1) run the notebook end-to-end, (2) checkout to the state before any models are trained, (3) rerun to the end of the notebook (thus creating a second branch), and measure the time taken to switch back to the first branch containing different models and plots.

We report the results in Fig 15b. Similar to §7.5.1, Kishu performs sub-second branch switching on 4/6 notebooks by updating (only) models and plots differing between branches (i.e., not the input dataframes) and does so up to 4.18× faster than the next best alternative (*StoreSales*). While there is considerable divergence between branches in the *StoreSales* test case (i.e., new auxiliary dataframes are created along ML models and plots), Kishu still performs branch switching at a fast 1.73 seconds, which is 4.18× faster than the next method (DumpSession). While Kishu+Det-replay can potentially be faster than Kishu (*TorchGPU*) by replaying cells that allow it to bypass expensive data loading, it can also cause unacceptable checkout times (**1050s**, *Cluster*, from replaying an entire deterministic model fitting sequence); hence, cost-based optimization is required for Kishu+Det-replay to function, which we leave to future work.

## 7.6 Fast Delta Detection

This section investigates Kishu's Co-variable granularity state tracking overhead by comparing the time taken by Kishu to track per-cell execution state delta with other tracking methods.

***Cumulative Tracking Overhead (Table 6).*** Kishu is consistently fastest at detecting state delta and is (1) up to 11.42× faster than the best out of IPyFlow and AblatedKishu (Check all) (*HW-LM*), and (2) only up to maximum of 2.03% of notebook runtime (*Sklearn*).

**Table 6: Kishu's delta tracking time vs. baselines. Kishu tracks the per-execution delta up to 11.42× faster than the next best method and is only up to 2.03% of notebook runtime.**

| | Tracking overhead (s) (% of notebook runtime) | | |
|---|---|---|---|
| **Notebook** | IPyFlow | AblatedKishu (Check all) | Kishu **(Ours)** |
| Cluster | 5.091s (0.299%) | 0.231s (0.014%) | **0.094s (0.005%)** |
| TPS | 15.49s (10.06%) | 4.618s (2.999%) | **1.195s (0.776%)** |
| Sklearn | 11.75s (2.293%) | 136.2s (26.61%) | **10.39s (2.030%)** |
| HW-LM | 30.07s (231.3%) | 2.964s (2.280%) | **0.259s (1.992%)** |
| StoreSales | **FAIL on cell 27** | 72.58s (10.91%) | **11.24s (1.69%)** |
| QiSkit | 18.91s (41.12%) | 2.235s (4.859%) | **0.339s (0.737%)** |
| Ray | 12.90s (0.546%) | 0.861s (0.036%) | **0.199s (0.008%)** |
| TorchGPU | 3.731s (0.521%) | 4.509s (0.630%) | **0.578s (0.081%)** |

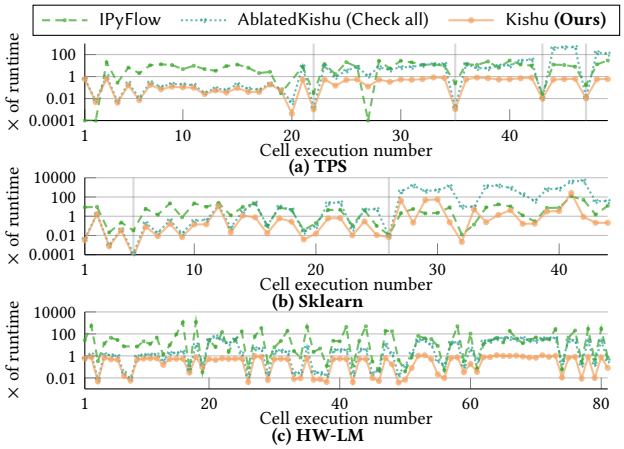


**Figure 16: Methods' per-cell tracking overhead as × of cell runtime. Gray vertical lines indicate long-running cells (>10s).** Kishu **outperforms baselines by efficiently detecting delta of long-running cells and identifying candidate updates (§4.3)**

***Per-Cell Tracking Overhead (Fig 16).*** We investigate per-cell execution tracking overhead of methods on selected notebooks. Kishu efficiently handles long-running cells (>10s, gray vertical lines) than IPyFlow: these cells often contain complex control flows (e.g., looped `if` statement, *Sklearn* cells 5 and 26[9]) and/or call complex functions (e.g., model fitting, all 4 long cells in *TPS*). As hypothesized in §2.4, IPyFlow incurs significant overhead on these cells (e.g., 0.3× on the 17s TPS cell 35) which Kishu circumvents by performing live analysis only between cell executions (0.3× → 0.001×).

Compared to AblatedKishu (Check all), Kishu identifies and only checks possibly updated Co-variables (§4.3): this exploits the incremental nature of cell executions (§2.2) and is *necessary*—in *Sklearn*, AblatedKishu (Check all)'s detection overhead grows significantly as more objects are introduced into the kernel (up to 4936×, cell 42), while Kishu's approach bounds the overhead (4936× → 0.84×).

Notably, there exists further optimization opportunities for Kishu such as for (1) cells updating Co-variables with nested VarGraphs (e.g. list of strings `text_neg`, *Sklearn* cell 41) and (2) read-only printing cells (e.g., `y_train[:10]`, *HW-LM* cell 67). Kishu incurs

[9] StoreSales cell 27 contains complex control flows; IPyFlow hangs indefinitely.

**Table 7: Variable vs. Co-variable count in notebooks.**

| Notebook | var# | Co-var# | Notebook | var# | Co-var# | Notebook | var# | Co-var# |
|---|---|---|---|---|---|---|---|---|
| Cluster | 28 | 26 | TPS | 47 | 47 | Sklearn | 43 | 42 |
| HW-LM | 172 | 170 | StoreSales | 27 | 27 | Qiskit | 51 | 41 |
| Ray | 29 | 28 | TorchGPU | 57 | 52 | | | |

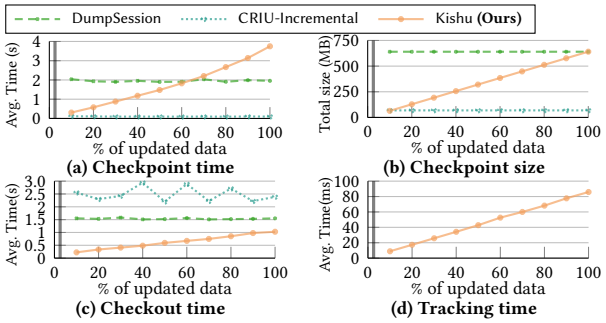


**Figure 17: Checkpoint/checkout efficiency vs. % of data in updated list Co-variable.** Kishu **performs best when each Co-variable contains few data in the state, which is typical of real-world notebooks (Table 7, vertical line marks average).**

significant overhead on these cases (260× and 1.06× respectively; while they are of low absolute value due to the short cell runtimes (547ms/2ms and 2ms/2ms respectively), this indicates need for more efficient delta detection methods such as list hashing and rule-based detection, respectively (§6.2), which we leave for future exploration.

## 7.7 Workload Study

This section studies Kishu's performance versus parameter sweeps on the degree of shared referencing between variables (i.e., Co-Variable size) (§7.7.1) and number of cell executions (§7.7.2).

*7.7.1 Performance vs. Shared Referencing.* We insert ten 64MB numpy arrays into a list (*% of state data in a Co-Variable*). We evaluate Kishu's checkpoint/checkout costs on cells modifying only *one* array in the list/Co-Variable. DumpSession's and CRIU-Incremental's performances are provided as comparison.

We report results in Fig 17. As Kishu detects updates and checkpoints at the Co-variable granularity, it checks for updates and then checkpoints all arrays in the list after each test cell. Kishu performs best when the list contains a low % of data in the state, as it can (1) limit the scope of update checking and (2) time-travel by saving/loading a small amount of state data. Kishu's performance drops as the list Co-variable bundles the changed array with more unchanged data and is equivalent to DumpSession[10] when all 10 arrays (i.e., all data in the state) form one large Co-Variable: Kishu has to (1) check the whole state for updates and (2) save/update all data for checkpoint/checkout, while CRIU-Incremental can still checkpoint only the one changed array in the (640MB) Co-variable.

However, as our evaluation workloads suggest (Table 7), states typically consist of a large number of small Co-Variables (each containing 2.57% of state data on average, black vertical lines in Fig 17). For these typical cases, our Co-Variable-based approach significantly outperforms other baselines (§7.3, §7.4).

[10]Different serialization libraries cause different checkpoint/checkout time cost; there is also overhead with managing data with blob storage vs. writing directly to file (§6.1).

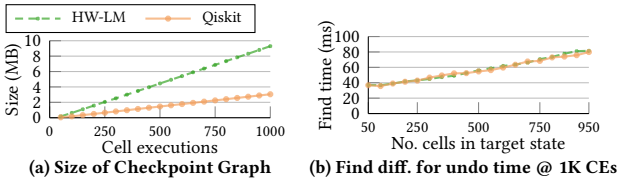


**Figure 18:** Kishu**'s scalability vs. cell executions. Checkpoint Graph size increases linearly; state difference computation time is linear vs. total cell count in current and target states.**

*7.7.2 Scalability to Long Notebook Sessions.* We choose 2 Data Visualization notebooks (*HW-LM* and *Qiskit*, and randomly re-execute up to 1000 cells.[11] We measure the (1) Checkpoint Graph size and (2) time to compute state difference for undoing 0-1000 cells (i.e., to a prior state) from the state after the 1000th cell execution.

We report results in Fig 18. The size of Kishu's Checkpoint Graph scales linearly with the number of executed cells and is up to only 9MB at 1000 cells (§7.7.2, *HW-LM*). Kishu's time to compute state difference between current and target states scales linearly with the *sum of cell count in the two states*, up to 81ms for any checkout operation in a 1000 cell execution-long session.[12] These overheads are negligible on a more typical notebook—only 133KB and 5.8ms, respectively, on our longest experiment notebook (*Qiskit*, 85 cells).

# 8 RELATED WORK

This section covers related work in PITR/checkpointing in various database applications (§8.1) and related notebook systems and their employed techniques for lineage tracing (§8.2). , and other applicable tools for saving/loading data in notebook states (§8.3)

## 8.1 PITR and Incremental Checkpointing

***PITR in Relational Databases.*** In many DBMSs, mechanisms like ARIES [101, 102] and its variants [12, 69, 105, 128, 129, 136, 153, 167] achieve durability and Point-in-time-Recovery (PITR) by combining (incremental) checkpointing and physical/operation logging [8, 71, 111, 129, 136, 152, 163, 167]. Persisting dirty objects (e.g., rows in relations) can enhance recovery efficiency by reducing the number of log entries to replay. ARIES [101] identifies dirty objects at the page level by recording a RecLSN, i.e., the earliest modification time, in the dirty page table. This is possible because all the pages are controlled by the transaction system's buffer manager. Kishu shares similarities: updates are periodically flushed. However, the core question is *how to define those updates* in computational notebooks. There are no buffer managers; moreover, variables/objects are dependent with inter-object references (i.e., memory pointers). Simply storing/restoring some variables independently thus can invalidate the state. Kishu addresses this with Co-Variables, atomic units of persistence that can correctly preserve inter-object dependencies. Kishu can still benefit from other PITR optimizations,

[11]This is the length of the longest observed notebook on Kaggle [68], and ~10 times the 97th percentile cell execution count of a notebook workload [127].

[12]This is because Kishu traverses the Checkpoint Graph to find the lowest common ancestor state on checkout with an off-the-shelf algorithm [159].

e.g., non-blocking checkpointing/restoration [101] and cost-based deterministic replay [108] (§7.5.2), which we leave as future work.

***PITR in Blob Storages.*** Blob storages have incorporated PITR for recovering its state to earlier points [4, 5, 13, 14, 70, 96, 108]. These systems typically log timestamped updates (e.g., Azure's change feeds [13] and SQLite's binary log [96]), which can be used to return the storage state to a previous snapshot. Computational notebooks require different approaches to determining state deltas, and Kishu addresses the unique challenge through Co-Variables and related techniques, which is the core novelty of this work.

***Incremental and Differential Checkpointing in HPC.*** HPC and stream processing systems implement efficient incremental checkpointing [10, 24, 84, 95]. These works focus on high-frequency (e.g., >1M/second) updates to relatively simple data structures (e.g., KV-stores [10, 95] and datasets [24, 84]). For enhanced efficiency, log truncation [95] and copying [84] are employed to ensure the bounded size and robustness of incremental logs. Computational notebooks require different delta detection methods as notebooks involve more complex objects, e.g., graphs, tensors, dataframes (§2.2). While orthogonal, the existing optimization techniques for HPC may offer significant performance benefits if the core techniques for Kishu are expanded to handle a significantly larger number of executions (e.g., scripts) compared to interactive notebooks.

## 8.2 Notebook Systems and Techniques

***Systems for Speeding up Data Exploration.*** There are a variety of works for enhancing data exploration efficiency in notebook-based systems [17, 22, 28, 73, 74, 86, 89, 90, 134, 160]. Reactive execution engines [86, 134, 160] track cell reruns and rerun their dependent cells reactively to enforce consistent cell outputs. Notebook recommender systems [89, 90] compute next-cell recommendations based on the current workflow. Symphony [17] and B2 [160] enable point-and-click interactions with ML models and dataframes, respectively, by translation to equivalent code operations. Burrito [73] and Vizier [22] construct and visualize a history graph for versioning multi-language (Python, R, SQL) data science code. Diff-in-the-loop [156] enables graphical comparisons of dataframes. Kishu facilitates data exploration through efficient time-traveling, and thus, is orthogonal and complementary to these works.

***Notebook State Versioning.*** Notebook state versioning has been explored by ForkIt [158], which performs backtracking and branching to/from states by saving entire states with pickle [55]. Our work versions notebook states in a much more efficient and robust manner compared to Forkit's approach (which is equivalent to our DumpSession baseline in §7) through efficient delta detection and incremental checkpointing and checkout.

***Lineage Tracing in Notebooks.*** Lineage tracing aims to capture code dependencies, i.e., accessed and updated data of cells, and has been widely used in notebook systems for a variety of downstream tasks (e.g., reactive execution) [3, 22, 38, 39, 86, 89, 90, 94, 98, 99, 134, 160]. Tracing methods can be divided into (1) static code analysis using methods like AST decomposition [50], and (2) live code instrumentation resolving variable/data references at runtime. They are relatively (1) cheap but conservative, and (2) accurate but expensive, respectively. Lineage tracers often combine static and live analysis, to mitigate conservative assumptions of static analysis [94] with dynamically computed 'ground truths' [39, 98, 134]. Unfortunately, to the best of our knowledge, all of these works, except for ElasticNotebook [94], detect modifications at the variable level, incorrectly disregard shared references between variables. Kishu's innovations in lineage tracing is efficient live analysis through (1) modeling the state at a coarser Co-variable granularity and (2) quickly pruning update candidates. This allows Kishu to achieve low false positives (§7.2) with low overhead (§7.6).

## 8.3 Checkpointing Notebook Objects

***Data Serialization for Checkpoint/Restore.*** Data in IPython-based (e.g., Jupyter) notebook session states can be saved with serialization libraries [32, 44, 53–55, 57, 100, 103], on which a variety of existing checkpointing tools are built: On-disk KV-stores save individual variables [2, 27, 59, 123, 135, 146], DumpSession [58] saves the session state in bulk, and ElasticNotebook [94] combines data storage/loading with cell replay for optimized session replication. These works do not checkpoint nor restore incrementally, or have limitations/require significant user effort: Dill and ElasticNotebook's checkpoint files must be entirely loaded for restoration; while KV stores can store and load parts of a state, delta detection and shared reference preservation must be manually handled. In comparison, Kishu can perform low-overhead incremental checkpointing and checkout while preserving shared references (i.e., correctness) and works with almost all data science libraries (§7.2).

***Memory Snapshotting.*** There exist OS-level checkpointing tools that can incrementally checkpoint a process for later restoration [7, 26, 35, 60, 79, 91, 117]. These tools identify and store dirty memory pages, then piece together the process image for restoration [35]: they often (1) cause large checkpoint sizes due to coarse page-level deltas [46], (2) can only handle single processes [36], and (3) can only restore from scratch: while we found a patent [107] and paper [49] enabling incremental checkpointing for multiprocessing jobs and with sub-memory-page granularity, respectively, we could not locate working implementations. In comparison, Kishu achieves lower checkpoint overheads via the logical Co-variable granularity deltas (§7.3), checkpoint multiprocessing notebooks via application-level instructions (§7.2), and incrementally restores via state difference detection and using existing kernel data (§7.5).

# 9 CONCLUSION

We have proposed Kishu, a new computational notebook system that offers efficient and fault-tolerant time-traveling between notebook states. Kishu captures session state evolution at a novel Co-variable granularity for efficient incremental checkpointing of state deltas, which Kishu then uses to perform incremental checkout with minimal data loading. Kishu's contributions include (1) low-overhead state delta detection, (2) branch-based state versioning, and (3) generalizable time-traveling—preserving inter-variable dependencies and handling missing data with fallback recomputation. We have shown that Kishu is compatible with 146 data science object classes and reduces checkpoint storage size and checkout time by up to 4.55× and 8.18×, respectively, on real-world notebooks.

# REFERENCES


[1] [n.d.]. CRIU CUDA Support. https://criu.org/What_cannot_be_checkpointed#Devices.

[2] 0xnurl. 2024. shelve — Python object persistence. https://github.com/0xnurl/redis-shelve.

[3] Raza Ahmad, Naga Nithin Manne, and Tanu Malik. 2022. Reproducible Notebook Containers using Application Virtualization. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, 1–10.

[4] Amazon. 2024. How Amazon Aurora helps you protect your data from mistakes. https://d1.awsstatic.com/events/reinvent/2020/How_Amazon_Aurora_helps_you_protect_your_data_from_mistakes_DAT403.pdf.

[5] Amazon. 2024. Streamlining Point-in-Time Recovery (PITR) for Amazon Aurora with AWS Backup. https://aws.amazon.com/blogs/storage/streamlining-point-in-time-recovery-pitr-for-amazon-aurora-with-aws-backup/.

[6] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[7] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–12.

[8] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant time recovery in Azure SQL database. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2143–2154.

[9] Anyscale. 2024. Ray - Effortlessly Scale Your Most Complex Workloads. https://www.ray.io/.

[10] Apache. 2024. Managing Large State in Apache Flink: An Intro to Incremental Checkpointing. https://flink.apache.org/2018/01/30/managing-large-state-in-apache-flink-an-intro-to-incremental-checkpointing/.

[11] Apache Arrow. 2024. PyArrow - Apache Arrow Python bindings. https://arrow.apache.org/docs/python/index.html.

[12] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[13] Microsoft Azure. 2024. Change feed support in Azure Blob Storage. https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-change-feed?source=recommendations&tabs=azure-portal.

[14] Microsoft Azure. 2024. Point-in-time restore for block blobs. https://learn.microsoft.com/en-us/azure/storage/blobs/point-in-time-restore-overview?source=recommendations.

[15] babreu ncsa. 2023. NCSA Qiskit Demo May 2023. https://github.com/babreu-ncsa/qiskit/blob/main/demo/QiskitDemo_NCSA_May2023.ipynb.

[16] Earl T Barr and Mark Marron. 2014. Tardis: Affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices* 49, 10 (2014), 67–82.

[17] Alex Bäuerle, Ángel Alexander Cabrera, Fred Hohman, Megan Maher, David Koski, Xavier Suau, Titus Barik, and Dominik Moritz. 2022. Symphony: Composing interactive interfaces for machine learning. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–14.

[18] Ekrem Bayar. 2022. Store Sales TS Forecasting - A Comprehensive Guide. https://www.kaggle.com/code/ekrembayar/store-sales-ts-forecasting-a-comprehensive-guide/notebook.

[19] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[20] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[21] Bokeh. 2024. bokeh.figure. https://docs.bokeh.org/en/latest/docs/reference/plotting/figure.html.

[22] Mike Brachmann, Carlos Bautista, Sonia Castelo, Su Feng, Juliana Freire, Boris Glavic, Oliver Kennedy, Heiko Müeller, Rémi Rampin, William Spoth, et al. 2019. Data debugging and exploration with vizier. In *Proceedings of the 2019 International Conference on Management of Data*. 1877–1880.

[23] Michael Brachmann and William Spoth. 2020. Your notebook is not crumby enough, REPLace it. In *Conference on Innovative Data Systems Research (CIDR)*.

[24] Greg Bronevetsky, Daniel J Marques, Keshav K Pingali, Radu Rugina, and Sally A McKee. 2008. Compiler-enhanced incremental checkpointing for openmp applications. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 275–276.

[25] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. 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*. 1–12.

[26] Yuqun Chen, James S Plank, and Kai Li. 1997. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*. 1–11.

[27] chest. 2024. chest - Simple on-disk dictionary. https://pypi.org/project/chest/.

[28] Supawit Chockchowwat, Zhaoheng Li, and Yongjoo Park. 2023. Transactional python for durable machine learning: Vision, challenges, and feasibility. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*. 1–5.

[29] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2022. Automatically finding optimal index structure. *arXiv preprint arXiv:2208.03823* (2022).

[30] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. Airindex: versatile index tuning through data and storage. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[31] Supawit Chockchowwat, Chaitanya Sood, and Yongjoo Park. 2022. Airphant: Cloud-oriented document indexing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1368–1381.

[32] cloudpipe. 2024. CloudPickle. https://github.com/cloudpipe/cloudpickle.

[33] Optuna contributors. 2024. Optuna: A hyperparameter optimization framework. https://optuna.readthedocs.io/en/stable/.

[34] Torch Contributors. 2024. torchvision - Image Transformers. https://pytorch.org/vision/stable/index.html.

[35] CRIU. 2023. Linux CRIU. https://criu.org/Main_Page.

[36] CRIU. 2024. CRIU - What cannot be checkpointed. https://criu.org/What_cannot_be_checkpointed.

[37] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2024–2027.

[38] Renato LF Cunha, Lucas C Villa Real, Renan Souza, Bruno Silva, and Marco AS Netto. 2021. Context-aware Execution Migration Tool for Data Science Jupyter Notebooks on Hybrid Clouds. In *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 30–39.

[39] Nachiket Deo, Boris Glavic, and Oliver Kennedy. 2022. Runtime provenance refinement for notebooks. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*. 1–4.

[40] The Devastator. 2023. Bruteforce Clustering. https://www.kaggle.com/code/thedevastator/bruteforce-clustering.

[41] Photutils Developers. 2024. An Astropy Package for Photometry. https://photutils.readthedocs.io/en/stable/.

[42] Matplotlib development team. 2024. Matplotlib - Figure. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.figure.html.

[43] Cody Dunne, Nathalie Henry Riche, Bongshin Lee, Ronald Metoyer, and George Robertson. 2012. GraphTrail: Analyzing large multivariate, heterogeneous networks while supporting exploration history. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1663–1672.

[44] Clark DuVall. 2015. serpy: ridiculously fast object serialization. https://serpy.readthedocs.io/en/latest/.

[45] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1555–1569.

[46] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

[47] Lightning AI et al. 2018. PyTorch ModelCheckpoint. https://pytorch-lightning.readthedocs.io/en/stable/api/pytorch_lightning.callbacks.ModelCheckpoint.html.

[48] Hugging Face. 2024. Hugging Face - The AI community building the future. https://huggingface.co/.

[49] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. 2011. libhashckpt: hash-based incremental checkpointing using gpu's. In *European MPI Users' Group Meeting*. Springer, 272–281.

[50] Python Software Foundation. 2023. Python - AST. https://docs.python.org/3/library/ast.html.

[51] Python Software Foundation. 2023. Python - Generators. https://wiki.python.org/moin/Generators.

[52] Python Software Foundation. 2023. Python Hashlib. https://docs.python.org/3/library/hashlib.html.

[53] Python Software Foundation. 2023. Python JSON. https://docs.python.org/3/library/json.html.

[54] Python Software Foundation. 2023. Python Marshal. https://docs.python.org/3/library/marshal.html.

[55] Python Software Foundation. 2023. Python Pickle Documentation. https://docs.python.org/3/library/pickle.html.

[56] The Linux Foundation. 2024. PyTorch. https://pytorch.org/.

[57] The Uncertainty Quantification Foundation. 2023. Dill - PyPi. https://pypi.org/project/dill/.

[58] The Uncertainty Quantification Foundation. 2023. Dill dump session. https://dill.readthedocs.io/en/latest/dill.html.

[59] Zope Foundation. 2024. ZODB programming guide. https://zodb.org/en/latest/guide/index.html.

[60] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-restart support for CUDA's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 302–313.

[61] GDB. 2024. GDB - Running programs backward. https://sourceware.org/gdb/current/onlinedocs/gdb.html/Reverse-Execution.html.

[62] Aurélien Geron. 2023. Chapter 4 – Training Models. github.com/ageron/handson-ml3/blob/main/04_training_linear_models.ipynb.

[63] Git. 2024. Git - Commit Graph. https://git-scm.com/docs/commit-graph.

[64] Git. 2024. git –fast-version-control. https://git-scm.com/.

[65] David Goodger. 2014. Code like a pythonista: Idiomatic python. *Archived from the original on* 27 (2014).

[66] Google. 2023. Keras. https://keras.io/.

[67] Google. 2024. Google Colab FAQ - Memory size. https://research.google.com/colaboratory/faq.html#available-memory.

[68] Google and X. 2022. Google AI4Code – Understand Code in Python Notebooks. https://www.kaggle.com/competitions/AI4Code.

[69] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The recovery manager of the System R database manager. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 223–242.

[70] The PostgreSQL Global Development Group. 1996. PostgreSQL - Continuous Archiving and Point-in-Time Recovery (PITR). https://www.postgresql.org/docs/current/continuous-archiving.html.

[71] The PostgreSQL Global Development Group. 1996. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[72] Google Groups. 2024. Time Travel Analysis or Undo in Jupyter. https://groups.google.com/g/jupyter/c/hMPDL7Iw_BQ/m/MWYv1d5cAwAJ.

[73] Philip J Guo and Margo I Seltzer. 2012. Burrito: Wrapping your lab notebook in computational infrastructure. (2012).

[74] Hari Sundaram Hanxi Fang, Supawit Chockchowwat and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+Data Space Version- ing. *CHI Conference on Human Factors in Computing Systems (CHI '25)* (2025). https://doi.org/10.1145/3706598.3714141

[75] HuggingFace. 2024. HuggingFace - BERT Tokenizer. https://huggingface.co/docs/transformers/en/main_classes/tokenizer.

[76] HuggingFace. 2024. HuggingFace - Pipelines. https://huggingface.co/docs/transformers/en/main_classes/pipelines.

[77] IPython. 2024. IPython Class. https://ipython.org/ipython-doc/3/api/generated/IPython.html.

[78] IPython. 2024. IPython Events. https://ipython.readthedocs.io/en/stable/config/callbacks.html.

[79] Twinkle Jain and Gene Cooperman. 2020. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[80] Jeremiah W Johnson. 2020. Benefits and pitfalls of jupyter notebooks in the classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education*. 32–37.

[81] Project Jupyter. 2023. Jupyter Notebook. https://jupyter.org/.

[82] Mario Juric, Steven Stetzler, and Colin T Slater. 2021. Checkpoint, Restore, and Live Migration for Science Platforms. *arXiv preprint arXiv:2101.05782* (2021).

[83] Kaggle. 2024. Kaggle Session Memory Size. https://www.kaggle.com/discussions/questions-and-answers/405504.

[84] Kai Keller and Leonardo Bautista-Gomez. 2019. Application-level differential checkpointing for HPC applications with dynamic datasets. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 52–61.

[85] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. 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*. 1–11.

[86] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[87] Tim Kraska. 2021. Northstar: An interactive data science system. (2021).

[88] ROUNAK KUMBHAKAR. 2024. pytorch resnet34 96.6 accuracy. https://www.kaggle.com/code/rounakkumbhakar/pytorch-resnet34-96-6-accuracy/notebook.

[89] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A Hearst, et al. 2021. Lux: always-on visualization recommendations for exploratory dataframe workflows. *arXiv preprint arXiv:2105.00121* (2021).

[90] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. 2023. Edassistant: Supporting exploratory data analysis in computational notebooks with in situ code search and recommendation. *ACM Transactions on Interactive Intelligent Systems* 13, 1 (2023), 1–27.

[91] Yawei Li and Zhiling Lan. 2010. FREM: A fast restart mechanism for general checkpoint/restart. *IEEE Trans. Comput.* 60, 5 (2010), 639–652.

[92] Zhaoheng Li, Supawit Chockchowwat, Hanxi Fang, Ribhav Sahu, Sumay Thakurdesai, Kantanat Pridaphatrakun, and Yongjoo Park. 2024. Demonstration of ElasticNotebook: Migrating Live Computational Notebook States. In *Companion of the 2024 International Conference on Management of Data*. 540–543.

[93] Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, and Yongjoo Park. 2024. Kishu: Time-Traveling for Computational Notebooks (Technical Report). *arXiv preprint arXiv:2406.13856* (2024).

[94] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *Proc. VLDB Endow.* 17, 2 (oct 2023), 119–133. https://doi.org/10.14778/3626292.3626296

[95] Chia-Yu Lin, Li-Chun Wang, and Shu-Ping Chang. 2020. Incremental checkpointing for fault-tolerant stream processing systems: A data structure approach. *IEEE Transactions on Emerging Topics in Computing* 10, 1 (2020), 124–136.

[96] LITEREPLICA. 2024. SQLite - Point in time recovery. https://litereplica.io/sqlite-point-in-time-recovery.html.

[97] Steven Loria. 2024. TextBlob: Simplified Text Processing. https://textblob.readthedocs.io/en/dev/.

[98] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2020. Fine-grained lineage for safer notebook interactions. *arXiv preprint arXiv:2012.06981* (2020).

[99] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: Multiversion Replay with Ordered Checkpoints. *arXiv preprint arXiv:2202.08429* (2022).

[100] MessagePack. 2024. MessagePack. https://github.com/msgpack.

[101] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[102] C Mohan and K Rothermel. 1988. Recovery protocol for nested transactions using writeahead logging. *IBM Tech. Dwclosure Bull. 31, 4 (Sept 1988)* (1988).

[103] Inc. MongoDB. 2023. BSON. https://pymongo.readthedocs.io/en/stable/api/bson/index.html.

[104] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[105] CB Morrey and Dirk Grunwald. 2003. Peabody: The time travelling disk. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings*. IEEE, 241–253.

[106] Andreas Mueller. 2024. WordCloud for Python documentation. https://amueller.github.io/word_cloud/.

[107] Michael Oliver Neary, Ashwani Wason, Shvetima Gulati, and Fabrice Ferval. 2007. Method and system for providing transparent incremental and multiprocess checkpointing to computer applications. US Patent 7,293,200.

[108] IBM Netezza. 2024. Non-deterministic SQL. https://www.ibm.com/docs/en/netezza?topic=environment-non-deterministic-sql.

[109] Inc. NumFOCUS. 2023. Pandas. https://pandas.pydata.org/docs/index.html.

[110] Inc. NumFOCUS. 2024. Pandas - DataFrame. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.

[111] Oracle. 2024. MySQL. https://www.mysql.com/.

[112] Oracle. 2024. MySQL - Point-in-Time (Incremental) Recovery. https://dev.mysql.com/doc/refman/8.0/en/point-in-time-recovery.html.

[113] Jim Ormond. 2018. ACM Recognizes Innovators Who Have Shaped the Digital Revolution.

[114] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[115] Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2013. Expositor: Scriptable time-travel debugging with first-class traces. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 352–361.

[116] photutils. 2024. photutils - ImageDepth. https://photutils.readthedocs.io/en/stable/api/photutils.utils.ImageDepth.html.

[117] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1994. *Libckpt: Transparent checkpointing under unix*. Computer Science Department.

[118] Plotly. 2024. Plotly - Low-Code Python Data Apps. https://plotly.com/.

[119] Polars. 2024. Polars - DataFrames for the new era. https://pola.rs/.

[120] Polars. 2024. Polars - LazyFrame. https://docs.pola.rs/py-polars/html/reference/lazyframe/index.html.

[121] PBC Posit Software, PBC formerly RStudio. 2023. Posit RStudio. https://posit.co/.

[122] NLTK Project. 2024. NLTK - Natural Language Toolkit. https://www.nltk.org/.

[123] Python. 2024. shelve — Python object persistence. https://docs.python.org/3/library/shelve.html.

[124] PyTorch. 2024. torch.tensor. https://pytorch.org/docs/stable/tensors.html.

[125] Qiskit. 2024. Qiskit - An open-source SDK for working with quantum computers at the level of extended quantum circuits, operators, and primitives. https://pypi.org/project/qiskit/.

[126] ray project. 2024. Overview of Ray. https://github.com/ray-project/ray-educational-materials/blob/main/Introductory_modules/Overview_of_Ray.ipynb.

[127] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[128] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing memory-resident databases. In *ICDE*. 452–462.

[129] Maximilian E Schule, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in main-memory database systems: From musaeusdb to tardisdb. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*. 169–180.

[130] scikit learn. 2024. scikit-learn - Machine Learning in Python. https://scikit-learn.org/stable/.

[131] scikit-learn developers. 2024. sklearn - GaussianMixture. https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html.

[132] scikit-learn intelex. 2024. Intel(R) Extension for Scikit-learn is a seamless way to speed up your Scikit-learn application. https://pypi.org/project/scikit-learn-intelex/.

[133] SciPy. 2024. SciPy - Fundamental algorithms for scientific computing in Python. https://scipy.org/.

[134] Shreya Shankar, Stephen Macke, Sarah Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, compact, and rapid program slicing for notebooks. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4038–4047.

[135] shove. 2024. shove - https://pypi.org/project/shove/. https://pypi.org/project/shove/.

[136] Emad Soroush and Magdalena Balazinska. 2013. Time travel in a scientific array database. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 98–109.

[137] Apache Spark. 2024. PySpark Documentation. https://spark.apache.org/docs/3.3.1/api/python/index.html.

[138] Apache Spark. 2024. pyspark.sql module. https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html.

[139] SQLite. 2024. SQLite. https://www.sqlite.org/.

[140] StackOverflow. 2024. Undo Pandas Dataframe Column Drop - StackOverflow. https://stackoverflow.com/questions/54284994/how-to-get-columnsseries-back-from-dropped-table.

[141] statsmodels developers. 2024. statsmodels - statistical models, hypothesis tests, and data exploration. https://www.statsmodels.org/stable/index.html.

[142] NumPy Team. 2024. NumPy - the fundamental package for scientific computing with Python. https://numpy.org/.

[143] Ray Team. 2024. ray.data.Dataset. https://docs.ray.io/en/latest/data/api/doc/ray.data.Dataset.html.

[144] The IPython Development Team. 2023. IPython Interactive Computing. https://ipython.org/.

[145] The IPython Development Team. 2023. Jupyter checkpoint. https://jupyter-server.readthedocs.io/en/latest/developers/contents.html.

[146] The IPython Development Team. 2023. Jupyter store magic. https://ipython.readthedocs.io/en/stable/config/extensions/storemagic.html.

[147] The Matplotlib Development Team. 2023. Matplotlib. https://matplotlib.org/.

[148] TensorFlow. 2024. TensorFlow - An end-to-end platform for machine learning. https://www.tensorflow.org/.

[149] TensorFlow. 2024. tf.Tensor. https://www.tensorflow.org/api_docs/python/tf/Tensor.

[150] transformers. 2024. transformers - State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow. https://pypi.org/project/transformers/.

[151] Cornell University. 2021. SKLearn Tweet Classification. https://github.com/CornellCAC/CVW_PyDataSci2/blob/master/code/sklearn_tweet_classification.ipynb.

[152] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[153] Joost SM Verhofstad. 1978. Recovery techniques for database systems. *ACM Computing Surveys (CSUR)* 10, 2 (1978), 167–195.

[154] Devlikamov Vlad. 2022. [TPS-Mar] Fast workflow using scikit-learn-intelex. https://www.kaggle.com/code/lordozvlad/tps-mar-fast-workflow-using-scikit-learn-intelex/notebook.

[155] Eric-Jan Wagenmakers and Simon Farrell. 2004. AIC model selection using Akaike weights. *Psychonomic bulletin & review* 11, 1 (2004), 192–196.

[156] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the loop: Supporting data comparison in exploratory data analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–10.

[157] Michael Waskom. 2024. seaborn: statistical data visualization. https://seaborn.pydata.org/.

[158] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork it: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[159] Wikipedia. 2024. Lowest Common Ancestor. https://en.wikipedia.org/wiki/Lowest_common_ancestor.

[160] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging code and interactive visualization in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 152–165.

[161] xgboost developers. 2024. XGBoost Documentation. https://xgboost.readthedocs.io/en/stable/.

[162] Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya G Parameswaran. 2021. Enhancing the interactivity of dataframe queries by leveraging think time. *arXiv preprint arXiv:2103.02145* (2021).

[163] Liqi Xu, Silu Huang, SiLi Hui, Aaron J Elmore, and Aditya Parameswaran. 2017. Orpheusdb: a lightweight approach to relational dataset versioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1655–1658.

[164] xxHash. 2023. xxHash - Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash.

[165] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[166] Emanuel Zgraggen, Robert Zeleznik, and Steven M Drucker. 2014. PanoramicData: Data analysis through pen & touch. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2112–2121.

[167] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 465–477.