Chipmink: Efficient Delta Identification for Massive Object Graphs

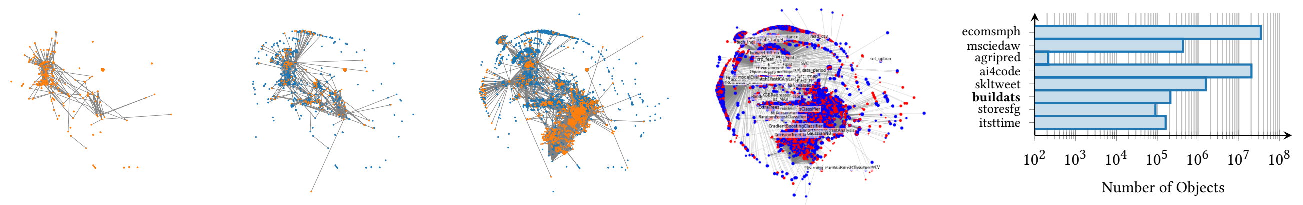
Supawit Chockchowwat
UIUC
supawit2@illinois.edu

Sumay Thakurdesai
UIUC
sumayst2@illinois.edu

Zhaoheng Li
UIUC
zl20@illinois.edu

Matthew S. Krafczyk
UIUC
mkrafcz2@illinois.edu

Yongjoo Park
UIUC
yongjoo@illinois.edu



(a) Import libraries **(b) Load & clean data** **(c) Train & test models** **(d) All objects and refs.** **(e) Object counts on real notebooks**

Figure 1: Underneath data systems lie a massive evolving object graph. Fig 1a to 1c illustrate object graph snapshots of `buildats` notebook with changed (orange) and unchanged (blue) objects, upto tens of millions of objects in real notebooks (Fig 1e).

ABSTRACT

Ranging from batch scripts to computational notebooks, modern data science tools rely on massive and evolving object graphs that represent structured data, models, plots, and more. Persisting these objects is critical, not only to enhance system robustness against unexpected failures but also to support continuous, non-linear data exploration via versioning. Existing object persistence mechanisms (e.g., Pickle, Dill) rely on complete snapshotting, often redundantly storing unchanged objects during execution and exploration, resulting in significant inefficiency in both time and storage. Unlike DBMSs, data science systems lack centralized buffer managers that track dirty objects. Worse, object states span various locations such as memory heaps, shared memory, GPUs, and remote machines, making dirty object identification fundamentally more challenging.

In this work, we propose a graph-based object store, named Chipmink, that acts like the centralized buffer manager. Unlike static pages in DBMSs, persistence units in Chipmink are dynamically induced by partitioning objects into appropriate subgroups (called ***pods***), minimizing expected persistence costs based on object sizes and reference structure. These pods effectively isolate dirty objects, enabling efficient partial persistence. Our experiments show that Chipmink is general, supporting libraries that rely on shared memory, GPUs, and remote objects. Moreover, Chipmink achieves up to 36.5× smaller storage sizes and 12.4× faster persistence than the best baselines in real-world notebooks and scripts.

PVLDB Reference Format:
Supawit Chockchowwat, Sumay Thakurdesai, Zhaoheng Li, Matthew S. Krafczyk, and Yongjoo Park. Chipmink: Efficient Delta Identification for Massive Object Graphs. PVLDB, 19(4): 603 - 616, 2025.
doi:10.14778/3785297.3785303

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

1 INTRODUCTION

Modern data science tools, ranging from batch scripts [38, 92, 93] to computational notebooks [12, 13, 53, 55, 58–60, 75, 101], rely upon a massive and evolving object graph—a new and increasingly critical modality of data. These underlying objects are used to represent many different forms of data, such as (semi-)structured data (e.g., `pandas`, `ElementTree`), graphs (e.g., `networkx`), machine learning models (e.g., `pytorch`, `tensorflow`), time series data, and so on. With the ability to save these objects efficiently and restore them correctly, we can improve systems' resilience against unexpected failures [21, 32, 98] and even allow data scientists to keep track of their work and explore alternative hypotheses [21, 35, 36].

However, existing persistence solutions are vastly infeasible. For each saving, existing persistence libraries like Pickle [40], Dill [66, 67], and ZODB [43] traverse these objects through their references and create a complete byte-representation snapshot of the object graph. This means saving variables at multiple points during a script's execution may capture the same objects, even if they remain unchanged (e.g., blue nodes in Fig 1a to 1c). As a result, these methods must process *2.96M* objects to save 42 snapshots of a real notebook `buildats` (Fig 2), incuring an infeasible storage and time cost prohibiting safer data exploration.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785303

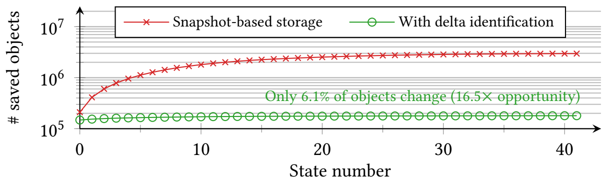


Figure 2: Cumulative numbers of `buildats`'s objects to be saved by snapshot-based storage vs. delta storage.

At the core, unlike DBMS'es [48, 51, 62, 68, 69, 91], the existing object persistence mechanisms lack *delta identification* (i.e., identifying differences between object graphs) to avoid unnecessary saving. Nonetheless, existing delta identifications only offer inaccurate, incomprehensive, and/or inefficient solutions. Graph databases like Neo4j [15] may inaccurately record nodes and edges after implicit updates where an execution appears static or access only a few variables but mutates nested shared structures. Manual approaches such as annotating objects to track [41, 52, 65, 77], translating into database formats [8, 23, 56, 72, 74, 87], or selectively saving some object types [11, 34, 71, 73, 82, 94] do not comprehensively capture all objects, especially given the diversity of user-defined types and external library internals (e.g., Fig 1d shows 4 built-in, 63 external, and 37 user-defined variables). Finally, classical algorithms like graph alignment [103, 104], edit-distance algorithms [44, 70, 76], and graph partitioning [7] are inefficient for the millions of objects and references present in real sessions (Fig 1e).

Goldilocks granularity of deltas: a proposal and challenges. By contrasting these solutions, we propose a novel concept for efficiently identifying deltas between massive and evolving object graphs: instead of object-level deltas and graph-level snapshots, one should find the Goldilocks (i.e., "just right") granularity of deltas by partitioning the object graph into subgraphs. To identify dirty objects between two points in time, we compare the subgraphs; only non-matching subgraphs need to be persisted.

However, finding the right granularity comes with two new challenges: within-subgraph and cross-subgraph. ***(Within-subgraph)*** We must optimize object-subgraph assignments in terms of both node composition and subgraph size. The composition should be *stable*: an object graph should consist of nearly the same set of subgraphs compared to the previous version, to minimize new subgraphs that need to be saved. The sizes should strike a *balance*: small enough to isolate only the dirty objects, yet large enough to reduce per-subgraph management overheads. ***(Cross-subgraph)*** All references across subgraphs must be properly preserved when saving dirty subgraphs and restoring prior states.

Our approach. With novel techniques to tackle the within-subgraph and cross-subgraph challenges, we present an algorithm called *podding*: a structure-aware partitioning of object graphs into subgraphs called *pods*. To determine the optimal composition and size of pods, we formulate an optimization problem that aims to minimize expected persistence costs considering the properties of object graphs including both node-specific properties (e.g., object size) and structural attributes (e.g., fanout). This principled approach naturally avoids extreme cases such as splitting or bundling all the objects.

Second, pods are designed to correctly preserve cross-pod edges. Cross-pod references can be considered virtual addresses that are later resolved to appropriate local references when related objects need to be put together during restoration.

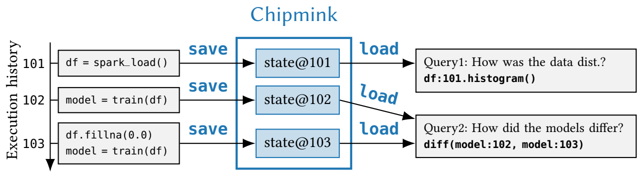


Figure 3: Chipmink saves and loads state for exploration.

Combining these techniques, we introduce Chipmink, a system designed to serve as an off-the-shelf persistence library. Aligning with the latest trends in data science and machine learning, Chipmink focuses on Python-based ecosystem, supporting 100% libraries empirically. This includes the libraries for data analytics (e.g., `pandas`, `arrow`), machine learning (e.g., `pytorch`, `tensorflow`, `scikit-learn`), distributed computing (e.g., `ray`), and data visualization (e.g., `matplotlib`, `seaborn`). As a result, Chipmink can fully replace existing generic persistence tools such as Pickle [40], Dill [66, 67], and Cloudpickle [25], delivering significant improvements in both time (up to 12.4×) and storage (up to 36.5×) efficiency.

2 MOTIVATION

Practical object persistence needs structure-aware deltas to reduce storage and time costs, otherwise infeasible through snapshotting and/or classical byte-string delta compression.

Limitation of snapshotting. Snapshotting (e.g., Dill and ZODB) stores and recovers a given state as a whole on disk. Suppose `dataset` is 10.1 GB large while `imputer` and `model` are 0.1 GB each, *snapshotting all three states* would take up 10.1 + (10.1 + 0.1 + 0.1) + (10.1 + 0.1 + 0.1) = *30.7 GB* space and proportional time to process them. Furthermore, if the user would like to "study the distribution of rows with missing values in `dataset` 9" (Query1) would require loading the entire 10.1 GB snapshot. That is, snapshotting can be inefficient for both saving and loading objects.

Limitation of byte-level deltas. Delta compression algorithms like LZ77 [105], xdelta [64], and others [49, 89, 99, 100] encode the differences of *target byte string* over *reference byte string(s)* [90]. For example, block move-based approaches [95] replace fixed-size blocks of byte-string deltas between the target and reference. While delta compression may reduce storage usage, it is slower than snapshotting because doing so requires both serializing the entire namespace and compressing the byte string each time. Meanwhile, loading involves reading both reference and target-delta byte strings, applying the delta, and deserializing the entire namespace. Determining the *reference namespace byte string* also remains an open question.

Our approach: structure-aware deltas. Instead, structure awareness can both reduce the storage and time overheads significantly. Suppose an object store detects that `imputer` only imputes 5% of the dataset (0.5 GB), the storage would only need to store the original dataset (10.1 GB), the imputed dataset (0.5 GB), `imputer` (0.1 GB), and two versions of `model` (0.1 GB each), totaling 10.1 + (0.5 + 0.1 + 0.1) + 0.1 = *10.9 GB*—almost a 3× reduction over snapshotting. Being aware of unrelated objects, the object store could filter out and bypass serializing those objects. Moreover, structure-aware deltas would allow precisely loading the relevant parts, e.g., when `model`

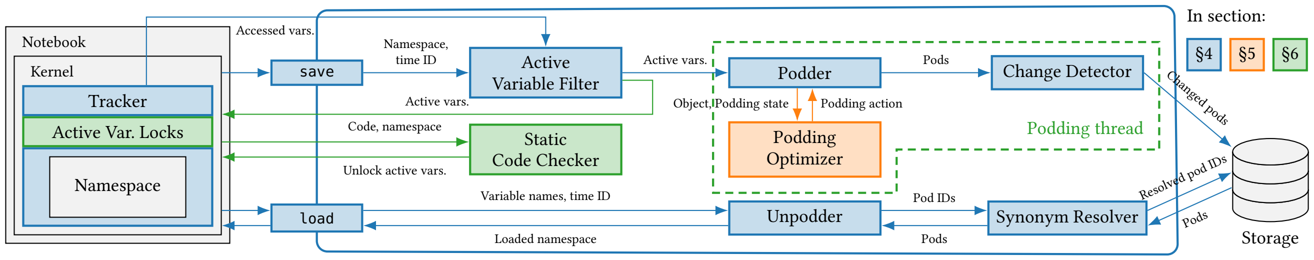


Figure 4: Chipmink's system architecture and interactions with computation notebook systems and the underlying storage. This data flow diagram illustrates Chipmink's partial saving and loading (§4, blue components), podding optimization (§5, orange components), and safe unblocking for uninterrupted exploration (§6, green components).

points to 25% of the dataset, answering Query1 and Query2 would load only 10.1 GB and (2.5 + 0.1) + (2.5 + 0.1) = 5.2 GB respectively.

3 SYSTEM OVERVIEW

This section presents interface (§3.1), system architecture (§3.2), data model (§3.3), and the equivalence guarantee (§3.4) of Chipmink to set the context before delving into technical details.

3.1 User APIs

With Chipmink, users can store and restore states through `save` and `load` APIs. When the user invokes **`save(namespace: NS) -> TimeID`** on a target namespace, they receive a `TimeID`. Later, the user can refer to this `TimeID` through **`load(names: set[str], time_id: TimeID) -> NS`** to restore the namespace. For example, knowing Chipmink, a user could save the states of the original `df:101` (in state@101) preprocessed through a Spark pipeline, an initial model `model:102` (in state@102), and an alternative model `model:103` (in state@103). Later on, the user is able to quickly load `df:101` to answer Query1 without re-executing the Spark pipeline. For Query2, the user can swiftly restore just `model:102` and `model:103` without loading the entire namespace.

3.2 System Architecture

Fig 4 illustrates Chipmink's architecture and interactions. Chipmink reads and writes to the runtime *kernels* (e.g., IPython's kernels for Jupyter). It also instruments namespaces for additional information and access control for safety. Chipmink stores its data in an *underlying storage* (e.g., key-value store). Chipmink comprises many components, enabling partial saving and loading, optimizing podding decisions, and safely unblocking exploration.

Partial saving and loading. Effective partial saving in Chipmink involves three steps: (1) identify the variables to save (Tracker and Active Variable Filter, §4.3), (2) decompose the variables' dependent objects into multiple pods (Podding, §4.1), and (3) detect the pods that have changed to be written to storage and link those unchanged with their synonymous pod IDs (Change Detector, §4.2). Inversely, Chipmink reverts the process to partially load a set of variable names: (1) resolve synonymous pods (if any) and read-only relevant pods from underlying storage (Synonym Resolver, §4.2), (2) assemble deserialized data from pods into original objects (Unpodinng, §4.1), and (3) send back the requested variables.

Intelligent podding optimization. Podding optimization instructs the podding process on how to decompose objects being saved. Given an object and the current podding state, the podding optimizer assesses different courses of action by calculating their anticipated costs and takes the best course of podding action. §5 presents the core of the optimizer: Learned Greedy Algorithm (LGA).

Concurrency for safe unblocking. Chipmink unblocks code executions as soon as possible and allows access to variables unrelated to the current saving. After identifying relevant variables being saved, it offloads the rest of the podding process into a *podding thread* (§6.1) and protects the relevant variables via *namespace locks* (§6.2) while allowing code executions to continue accessing other variables. It also automatically *checks for read-only code executions* (§6.3) to grant users free access to all variables.

3.3 Data Model

This section covers Chipmink's conceptual model for capturing interdependent objects and useful for discussions on partial saving and loading, object splitting, as well as asynchronous saving.

Namespace, objects, and variables. A *namespace* is a collection of *objects* and their *names*. These named objects are interchangeably called *variables*. Fig 5 shows an instance of a namespace, 5 variables, and 9 objects. An object u may *(directly) depend* on another object v if and only if object u has a reference to object v.

Object graph. ObjectGraph is a labeled and rooted directed graph $\mathcal{G} = (\mathcal{U}, \mathcal{E}, \mathcal{V}, \ell)$ where each node $u \in \mathcal{U}$ represents an object and a directed edge $e = (u, v) \in \mathcal{E}$ implies that object u directly depends on v. $\mathcal{V} \subseteq \mathcal{U}$ is the set of variables (named objects) whereas the variable naming function ℓ is a mapping from a string variable name to its variable $u \in \mathcal{V}$. ObjectGraph considers the namespace dictionary object [1] the *root* of the graph.

Code execution. A code execution Exec runs a code block C (e.g., Python statement(s)) on the current state $\mathcal{G}$ to create a new state: $\text{Exec}(\mathcal{G}, C) = \mathcal{G}^{+} = (\mathcal{U}^{+}, \mathcal{E}^{+}, \mathcal{V}^{+}, \ell^{+})$. Depending on the context, code execution is also interchangeably called "cell execution," "program execution," or simply "execution."

We presume the *code execution locality*: a code execution can only change the objects and dependencies connected to the variables accessed in the code block.[2] For example, code execution can change the "title" object of a figure by accessing and using any connected variable like `ax` or `plt` and *not* by accessing unconnected variables like `__name__`. Let $\mathcal{V}_C$ be the set of accessed variables in C, an

[1]The object returned from IPython kernel's `globals()`.
[2]Dismissing unsafe pointer arithmetics and dereferencing which are rare in data science programs and libraries.

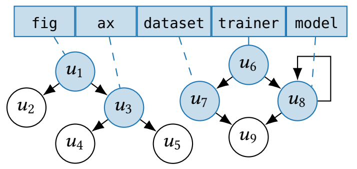


Figure 5: An instance of ObjectGraph. Namespace consists of objects $\mathcal{U}$ (circles) and dependencies $\mathcal{E}$ (arrows) between them. Each variable naming ℓ (blue dash line) connects a variable name (rectangle) to its object $v \in \mathcal{V}$ (blue circle).

object $u \in \mathcal{U}$ does not change (i.e., $u \in \mathcal{U}^+$) if the object u is not connected to any accessed variable $v \in \mathcal{V}_C$. Similarly, a dependency $e = (u, u') \in \mathcal{E}$ does not change $e = (u, u') \in \mathcal{E}^+$, if object u or u' is not connected to accessed variables $v \in \mathcal{V}_C$.

Pod dependency graph. As a result of podding, Chipmink produces a *pod dependency graph* or PodGraph $\mathcal{G}_p = (\mathcal{U}_p, \mathcal{E}_p)$ based on ObjectGraph. Here $\mathcal{U}_p$ is the set of pods each containing a disjoint partitioning set of objects from ObjectGraph $\mathcal{G} = (\mathcal{U}, \mathcal{E}, \mathcal{V}, \ell)$, i.e., $\forall u_p, v_p \in \mathcal{U}_p, u_p \neq v_p, u_p \cap v_p = \varnothing$ and $\bigcup_{u_p}^{\mathcal{U}_p} u_p = \mathcal{U}$. An edge $e_p = (u_p, v_p)$ exists in $\mathcal{E}_p$ if and only if an object in the source pod u_p has an edge to an object in the destination pod v_p, i.e., $\exists u \in u_p, \exists v \in v_p, (u, v) \in \mathcal{E}$. PodGraph is useful for prefetching dependent pod bytes during unpodding, identifying active variables (§4.3), and safely locking variables being saved (§6).

3.4 Equivalence Guarantee

Chipmink guarantees to store and load the same objects when their *serialized* representations are *equal*. Serialization `Ser` transforms an ObjectGraph $\mathcal{G}$ into a byte stream $\texttt{Ser}(\mathcal{G}) = \mathcal{B}$ that can written to storage. Conversely, *deserialization* `Deser` reverts the byte stream back to the ObjectGraph $\texttt{Deser}(\texttt{Ser}(\mathcal{G})) = \mathcal{G}$. Chipmink's guarantee based on serialized representations is consistent if deserialization is *consistent*: deserializing the same byte stream produces the same object $\texttt{Deser}(\mathcal{B}) = \texttt{Deser}(\mathcal{B}')$ if $\mathcal{B} = \mathcal{B}'$. In Python, Pickle [40] as well as its extensions like Dill [66, 67] and Cloudpickle [25] are examples of serialization protocols that correctly operate with ObjectGraph. Our notion of equality is similar to Li et al. [61] where both value and structural information are considered.

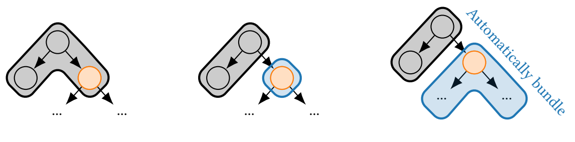
Serialization equality. Two ObjectGraph $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent under the *serialization equality* if and only if $\texttt{Ser}(\mathcal{G}_1) = \texttt{Ser}(\mathcal{G}_2)$. On the contrary, we define *dirty/changed/modified/mutated* ObjectGraph $\mathcal{G}$ as a ObjectGraph with a different serialized byte stream with respect to its previous state $\mathcal{G}'$: $\texttt{Ser}(\mathcal{G}) \neq \texttt{Ser}(\mathcal{G}')$.

4 ENABLING PARTIAL SAVING AND LOADING

Chipmink achieves a partial saving of modified objects and loading of variables through a novel concept called *podding* that groups objects into *pod(s)* (§4.1). Chipmink detects changes in the serialized bytes of pods over time and only saves changed pods (§4.2). To reduce overheads, Chipmink utilizes a novel filtering technique to identify variables accessed directly and indirectly (§4.3).

4.1 Podding and Unpodding

Podding. Chipmink leverages a novel concept named *podding* which partitions ObjectGraph's objects into multiple pods, each



(a) Bundle **(b) Split-continue** **(c) Split-final**

Figure 6: Different podding decision actions on the same current object (orange circle), parent pod (grey round box), and new pod (blue round box) if any.

producing its byte stream (a.k.a. *pod bytes*) via a serialization protocol, tagged with a *pod ID*. Each pod ID and pod bytes form a unit of data to be written to a storage. With ObjectGraph divided into pods, Chipmink can incrementally save the namespace by isolating and writing only changed pods. Furthermore, Chipmink can partially load by reading and unpickling only the necessary pods.

As the serialization protocol traverses through objects in ObjectGraph in depth-first order,[3] podding also observes the objects and consults *podding decision* on how to construct pods. On each object in ObjectGraph, the podding decision selects either one of three *podding actions* to (1) **bundle** the object with the current pod, or (2) **split** the object into a separate pod. In case of a split, the podding decision also decides whether to (a) **split-continue** which splits the object and recursively continues applying the podding decision on descendent objects, or (b) **split-final** which splits the object and skips further podding on descendent objects (Fig 6).

Chipmink's efficiency in incremental saving and partial loading hinges on a good podding decision. Naïvely splitting each object into separate pods introduces tremendous pod management overheads while using one pod for the whole graph reverts podding and disregards partial saving and loading opportunities (see §8.7). We will shortly derive an intelligent optimization to discover the optimal podding decision (§5).

Unpodding. As opposed to podding that splits objects, *unpodding* assembles objects back together during loading. Chipmink watches for relevant pod IDs in the byte stream. When one is found, Chipmink reads the associated pod bytes from storage and recursively deserializes the pod bytes.

Base serialization. Chipmink applies to any serialization protocol that supports (explicitly or implicitly) a mechanism to serialize dependent objects to multiple byte streams. Currently, we implement Chipmink to be fully compatible with Pickle [40] as well as its extensions like Dill [66, 67] and Cloudpickle [25].

Encoding references within and across pods. Chipmink needs a special protocol to encode shared references across pods because, for Pickle-like serializations, *memo IDs*—numbers denoting object references—must be natural numbers for references within pod and the total number of objects is unknown during serialization.

Chipmink implements a *virtual memo space* that denote references for serialized data using *virtual memo IDs* while uniquely associating object references to *global memo IDs*. In this protocol, virtual memo IDs for within-pod references are the original natural-number memo IDs, while those for across-pod references are their global memo IDs plus 2^{31}. To uniquely associate references to global memo IDs, each pod allocates page(s) of B global

[3]By following reduction path depending on the object type, e.g., Pickle's builtin save functions, `__getinitargs__`, `__getstate__`, `__reduce__`, etc.

memo IDs in range $[\delta_i, \delta_i + B)$ dynamically as needed. As the pod encounters a new object reference, it associates the reference to a global memo ID in the allocated page. Chipmink persists these page offsets $\{\delta_i\}$ as metadata. Therefore, given a virtual memo ID, Chipmink can retrieve the object reference stored at the global memo ID calculated by Equation (1).

$$m_{\text{global}}(m_{\text{virtual}}) = \begin{cases} \delta_i + r, & \text{if } m_{\text{virtual}} < 2^{31} \\ m_{\text{virtual}} - 2^{31}, & \text{if } m_{\text{virtual}} \geq 2^{31} \end{cases} \quad (1)$$

$$\text{where } i = \lfloor m_{\text{virtual}} \ / \ B \rfloor \text{ and } r = m_{\text{virtual}} \bmod B$$

4.2 Change Detector and Synonym Resolver

Chipmink aims to detect changes in pod bytes based on the serialization equality (§3.4). Changed pods signify new data to write while unchanged pods present an opportunity to save storage.

Detecting new pods. To detect new pods, Chipmink maintains a cache-like mapping structure in memory, named *pod thesaurus*, that associates each written pod bytes to its pod ID. Pod IDs that share exactly the same pod bytes are called synonymous pod IDs. Before passing pod bytes to the underlying storage, Chipmink checks whether the pod thesaurus is missing the pod bytes. If so, Chipmink writes the pod bytes to storage, then inserts new pod bytes and pod ID to the pod thesaurus. On insertion, the pod thesaurus may evict unused pod bytes and pod IDs until it fits within its capacity. We select the last in first out (LIFO) eviction policy for its simplicity.

Skipping and reading redundant pods. Otherwise, if the pod thesaurus already contains the pod bytes (i.e., it was unchanged from a prior write), Chipmink skips writing pod bytes and instead notes down the synonymous pod IDs in the storage. On reading a pod ID, if the pod ID is associated with a synonymous pod ID, Chipmink reads pod bytes of the synonymous pod ID instead.

Thesaurus of hashes for lower memory usage. In place of exact pod bytes, pod thesaurus can store the hashes of pod bytes with their pod IDs. To avoid collision with high probability, the pod thesaurus should hold only a limited number of pods while relying on a large enough number of hash bits. For example, in our experiment (§8), we use 128-bit `xxhash` and set the thesaurus capacity to 1 GB, meaning it can hold 62.5 M (1 GB / 16 B) pods. Even after 1 billion pods, such a capacity has a collision probability of only 1.8×10^{-22}. In the rare case of a collision, Chipmink would raise an error on load due to missing references and unpickling errors. Given the execution history, one can recover by loading the previous state and re-executing cells. Further safeguards (e.g., combining hashes with checksums or length checks) can reduce collision risk further.

4.3 Active Variable Filter

Variables not referred to during code execution presents an opportunity to reduce change detection overhead further. We call these unreferenced variables *inactive variables* and referenced variables *active variables.* A variable is active when it is connected in ObjectGraph to an accessed variable in the recent execution Exec($\mathcal{G}, C$). For example, in Fig 7, executing `model.predict(...)` accesses the variable `model` so `dataset`, `trainer`, and `model` are active variables, while `fig` and `ax` are inactive variables based on the ObjectGraph.

Based on §3.3, Chipmink can identify inactive and active variables by tracking namespace accesses and expanding from accessed variables to active variables using PodGraph. Theorem 4.1 shows that this mechanism is sufficient for determining active variables.

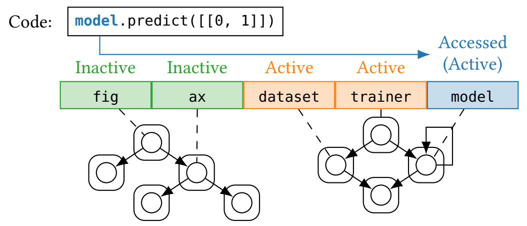


Figure 7: Inactive, active, and accessed variables.

THEOREM 4.1. *All active variables belong to pod(s) that are connected to accessed variables' pod(s) on the prior PodGraph.*

PROOF. Let u^{act} be any active variable, it is connected to an accessed variable u^{acc} in $\mathcal{G}$ at a point in time by definition. We use the following lemmas to prove the statement.

LEMMA 1. *By code execution locality (§3.3), if $u, u' \in \mathcal{U}$ are connected in $\mathcal{G}$, u is connected to $\exists\, v \in \mathcal{V}_C$ in the prior $\mathcal{G}^0$.*

LEMMA 2. *By pod dependency graph properties (§4.1), if $u, v \in \mathcal{U}$ are connected in $\mathcal{G}$, pods $u_p \ni u$ and $v_p \ni v$ are also connected in $\mathcal{G}_p$*

While the proofs of lemmas are deferred, key assumptions and properties give hints to derive them. By Lemma 1, u^{act} is connected to a $v^{\text{acc}} \in \mathcal{V}_C$ in the prior $\mathcal{G}^0$. By Lemma 2, it implies that $u_p \ni u^{\text{act}}$ and $v_p \ni v^{\text{acc}}$ are also connected in the prior PodGraph $\mathcal{G}_p^0$. □

5 PODDING OPTIMIZATION

For each object encountered during serialization, a podding optimizer chooses the best podding action (whether to bundle, split-continue, or split-final the object) to minimize storage cost. The podding optimizer must be designed as an *online streaming algorithm* to decide podding actions in one serialization pass per saving to *efficiently* and *generalizably* handle large numbers of objects of all types. This section proposes Chipmink's podding optimizer—Learned Greedy Algorithm (LGA)—by introducing an abstract objective function (§5.1), modeling related uncertainties (§5.2), formulating a podding problem (§5.3), and presenting the algorithm (§5.4).

5.1 High-level Objective

LGA aims to minimize the storage cost by partitioning objects $\mathcal{U}$ into pods $\mathcal{U}_p$. Based on performance profiles, the total podding cost is dominated by (1) overheads per pod c_{pod}, including storage overheads and additional podding time, and (2) rewritten pods due to changed objects:

$$\mathrm{L}(\mathcal{U}_p; \mathcal{G}) = \sum_{u_p}^{\mathcal{U}_p} \left[c_{\text{pod}} + s(u_p)\ \Phi(u_p) \right] \quad (2)$$

where c_{pod} is a constant overhead per pod, $s(u_p)$ is the size of the pod u_p, and $\Phi(u_p)$ is the random-variable number of times the pod u_p changes (modeled in §5.2). Here we make simplifying assumptions that $s(u_p) = \sum_u^{u_p} s(u)$ and the size of an object $s(u)$ can be measured through a function (e.g., `sys.getsizeof` [4]).

[4]This function does not exactly measure but provides a signal sufficiently correlated with the size of the object after serialization.

5.2 Composable Volatility Model

To support LGA in weighing between podding actions, *volatility model* predicts object mutations and estimates the mutations of different poddings of objects. For these purposes, LGA represents object mutations as a Poisson distribution widely used to model event rates [24] $\mathrm{Pois}(\lambda(u))$ where the mutations occur $\lambda(u) \leq 1$ times per code execution (i.e., rate of changes). This volatility model guides LGA in separating volatile and non-volatile objects apart, reducing unnecessary pod changes. Moreover, LGA can hypothesize the volatility of a pod via Poisson's *composability* $\lambda(u_p) = \sum_u^{u_p} \lambda(u)$.

While object changes may exhibit correlations with themselves or across time (e.g., training vs. testing phases), our Poisson assumption is a simplifying choice to minimize model and optimization overheads. To address this limitation, future work may extend the model and the subsequent algorithm to account for correlated or higher-order statistics of object changes and study the resulting effect of improved volatility model on the optimality gap.

Parameter estimation. To process millions of objects, LGA estimates the per-object volatility $\lambda(u)$ based on lightweight and broadly applicable feature extraction to arbitrary object types. We initially profiled 14 such features (e.g., ID, scope, type-based functions, functional tests, different measures of size) and measured their importance against tracked object changes on a few notebook samples.[5]. The important features with high average gain [28] used to train the final model are *the object's immediate size, object's length (if any), and the length of the object's `__dict__`*. Note that more sophisticated domain-specific features (e.g., object role or code complexity) are possible extensions, but the current choice is pragmatic for generality. After training and validating many models, we find that gradient boosting methods such as XGBoost [22] and LightGBM [54] perform well with compact models and settle with LightGBM for its inference speed in our implementation.

5.3 Objective Function

With the composable volatility model, LGA substitutes $\Phi(u_p) = \mathrm{Pois}(\lambda(u))$ and optimizes the expectation of the probabilistic cost function, $\mathcal{L}(\mathcal{U}_p; \mathcal{G}) = \mathbb{E}[\mathrm{L}(\mathcal{U}_p; \mathcal{G})]$:

$$\mathcal{L}(\mathcal{U}_p; \mathcal{G}) = \sum_{u_p}^{\mathcal{U}_p} \left[c_{\text{pod}} + \sum_{u}^{u_p} s(u) \sum_{v}^{u_p} \lambda(v) \right] \tag{3}$$

Hardness. LGA's podding problem is NP-hard, via equivalences to a *tree partitioning* and subsequently a *supermodular minimization*, a dual to the NP-hard *submodular maximization* problem [37]. We defer the proof to our technical report.

5.4 Learned Greedy Algorithm

While it is possible to derive *two-pass approximate solution* from an approximation algorithm (e.g., RandomizedUSM [16]) for submodular maximization, the additional serialization pass would double the algorithm computation overhead and the algorithm would increase the implementation complexity. Chipmink settles on a one-pass greedy solution that discovers good podding solutions more quickly.

[5]These notebooks are held out and not benchmarked against in the experiments.

Algorithm 1: LGA decision, `lga_action`$(u, u_p, \mathcal{G}_p)$

```
Input: Object $u$, current pod $u_p$, current pod dependency graph $\mathcal{G}_p$
Output: Podding action $a \in$ {bundle, split-continue, split-final}
1 $\Delta\mathcal{L}_{\text{bundle}} \leftarrow s(u_p) \times \lambda(u) + s(u) \times (\lambda(u_p) + \lambda(u))$
2 $\Delta\mathcal{L}_{\text{split}} \leftarrow c_{\text{pod}} + s(u) \times \lambda(u)$
3 pod_depth ← PodDepth($u_p, \mathcal{G}_p$)
4 if $\Delta\mathcal{L}_{bundle} < \Delta\mathcal{L}_{split}$ then return bundle
5 else if pod_depth < MAX_POD_DEPTH then return split-continue
6 else return split-final
```

One-pass greedy solution. Our one-pass greedy solution, LGA, selects the podding action with the locally optimal cost. Given a target object and the current podding state, it measures and compares the additional cost to bundle $\Delta\mathcal{L}_{\text{bundle}}$ and the additional cost to split $\Delta\mathcal{L}_{\text{split}}$. If the former is lower, it decides to bundle the object. Otherwise, it decides to split-continue or split-final.

Bundling the object to the current pod makes the pod larger and more volatile. The differences $\Delta\mathcal{L}_{\text{bundle}}$ can be calculated as:

$$\begin{aligned} \Delta\mathcal{L}_{\text{bundle}} &= \mathcal{L}_{\text{bundle}} - \mathcal{L}(\mathcal{U}_p; \mathcal{G}) \\ \Delta\mathcal{L}_{\text{bundle}} &= [c_{\text{pod}} + (s(u_p) + s(u))(\lambda(u_p) + \lambda(u))] \\ &\quad - [c_{\text{pod}} + s(u_p)\lambda(u_p)] \\ \Delta\mathcal{L}_{\text{bundle}} &= s(u_p)\lambda(u) + s(u)(\lambda(u_p) + \lambda(u)) \end{aligned} \tag{4}$$

On the other hand, split actions maintain the current pod's cost at the expense of an additional pod overhead instead (Equation (5)).

$$\Delta\mathcal{L}_{\text{split}} = c_{\text{pod}} + s(u)\lambda(u) \tag{5}$$

LGA selects split-continue if the distance between the pod and the root pod is lower than a constant `MAX_POD_DEPTH`; otherwise, it selects split-final. In addition, LGA memoizes previous podding decisions for each object to regulate the composition across pods and to reduce redundant computation.

6 MINIMAL BLOCKING FOR EXPLORATION

While Chipmink must ensure the consistency of active variables and all their dependent objects during saving, it facilitates the user to continue exploring data in parallel by capitalizing on two key opportunities: when the exploration (1) accesses inactive variables, or (2) only statically reads active variables. Chipmink concurrently saves in a separate thread (§6.1) without locking inactive variables (§6.2) and safely permits executions of static code (§6.3).

6.1 Podding Thread for Asynchronous Saving

Chipmink employs thread-based parallelism to concurrently save active variables.[6] After identifying active variables, Chipmink initiates a new *podding thread* to proceed with the remaining saving steps, which include optimized podding, detecting changed pods, and writing them to the underlying storage (see Fig 4). Currently, Chipmink only allows a single concurrent save at a time; if the next saving is requested before the podding thread finishes, it will have to wait by joining the existing podding thread first.

[6]While standard Python currently executes one thread at a time due to the Global Interpreter Lock (GIL), PEP 703 [45]—a formally accepted enhancement—is making GIL optional, thereby enabling thread-based parallelism in the near future.

6.2 Active Variable Locking

Synchronization is necessary to protect active variables from being altered while Chipmink is saving them to avoid data corruption. Chipmink could opt to naively lock down the entire namespace and concurrently save active variables. Nonetheless, this approach mostly equivalently blocks subsequent execution except in rare cases when executions do not access the namespace.

Locking active variables. After identifying accessed variables (§4.3), Chipmink minimally locks *active variables* and leaves inactive variables free for access. Instead of applying as many locks as the number of active variables, two simple non-reentrant locks suffice: one locking namespace l_{ns} to make the namespace thread-safe and another locking active variables l_{active} held during saving.

6.3 Safe Static Execution

Apart from accessing inactive variables, users should be able to explore active variables if the exploration only reads those variables statically without modification. Such exploration is said to be a *static execution* over a *static code*.

Rather than laying the burden on users to declare static code accurately, we should automatically and safely check for static codes. In other languages such as Rust [85, 86] and C++ [30], one can determine the static code through reference annotations; however, this is neither supported nor common practice in dynamically typed languages like Python. Alternatively, we can instrument every object to watch for object mutations and acquire locks appropriately; nevertheless, such fine-grained instrumentation is excessively expensive to implement.

Allowlist-based static code checker. Chipmink thus checks for static code using allowlist-based static code checker (ASCC), which utilizes an allowlist of patterns in the abstract syntax tree (AST) that represents static code. Beyond syntactic information, ASCC also utilizes types presented in the namespace at runtime, allowing Chipmink to recognize more static codes. With an allowlist of rules, ASCC parses out the AST from a given code and traverses through AST nodes. At each node, ASCC searches for a rule in the allowlist that matches the node's subtree, e.g., subtrees corresponding to `print(x)`, `df.head()` (where `df` is a `DataFrame`), or `np.mean(arr)`. If a node does not match with any rule, the code is declared non-static.

While the allowlist is extensible by domain experts like the users (e.g., to include static user-defined functions), Chipmink prepopulates the allowlist with definitely static rules such as printing a string, calculating a summation `sum(...)`, and `DataFrame.head()`.

7 DISCUSSION

This section analyzes the Chipmink's correctness (§7.1), optimality (§7.2), and stability (§7.3), as well as discussing generalizability to other environments (§7.4) and implementation details (§7.5).

7.1 Correctness Guarantee

THEOREM 7.1. *Ser(Unpod(Pod($\mathcal{G}$))) = Ser($\mathcal{G}$), establishing that Chipmink 's podding–unpodding preserves the original object graph.*

PROOF. The serialized representation Ser($\mathcal{G}$) is consisted of (1) information *within* each pod and (2) information *across* pods. Let $\mathcal{G}'$ = Unpod(Pod($\mathcal{G}$)), for the within part, podding produces subgraphs of $\mathcal{G}$, and during unpodding, each subgraph in $\mathcal{G}'$ is obtained by deserializing the corresponding stored subgraph from $\mathcal{G}$; thus each subgraph in $\mathcal{G}'$ is serialized-equivalent to its counterpart in $\mathcal{G}$. For the across part, Chipmink records all inter-pod references exactly during podding and replays them during unpodding, ensuring that these references in $\mathcal{G}'$ match those in $\mathcal{G}$. Since both the internal content of each subgraph and the inter-subgraph references are identical in $\mathcal{G}'$ and $\mathcal{G}$, we have Ser($\mathcal{G}'$) = Ser($\mathcal{G}$), proving that Ser(Unpod(Pod($\mathcal{G}$))) = Ser($\mathcal{G}$). □

7.2 Approximate Optimality

THEOREM 7.2. *Let $\mathcal{L}^{LGA}$ be LGA's cost and L^* be the optimal cost, $\mathcal{L}^{LGA} \leq (1+\alpha)\,\mathcal{L}^*$ where $\alpha = \min\left\{c_{pod}/(2\gamma), \sqrt{c_{pod}/(16\mu_s\mu_\lambda)}\right\}$ is defined by average sized volatility $\gamma = \frac{1}{|\mathcal{U}|}\sum_u s(u)\lambda(u)$, average size $\mu_s = \frac{1}{n}\sum_u s(u)$, and average volatility $\mu_\lambda = \frac{1}{|\mathcal{U}|}\sum_u \lambda(u)$.*

PROOF. The relative error is derived from the ratio of a lower bound of the optimal cost $\mathcal{L}^*$ and a upper of bound LGA cost $\mathcal{L}^{\text{LGA}}$. For a non-empty graph, any podding solution must incur at least $c_{\text{pod}} + \sum_u s(u)\lambda(u) \leq \mathcal{L}^* \leq \mathcal{L}^{\text{LGA}}$. A upper bound is derived from Lemma 3, similarly to DeterministicUSM's approximation [16].

LEMMA 3. *LGA has a worst case cost: $\mathcal{L}^{LGA} \leq \frac{1}{2}(\mathcal{L}^* + \mathcal{L}^{split})$.*

The deferred proof considers the minimization of the submodular dual $f = \mathcal{L}^{\text{split}} - \mathcal{L} \geq 0$ where $\mathcal{L}^{\text{split}} = |\mathcal{U}|(c_{\text{pod}} + \gamma)$. □

7.3 Podding Stability

We define *podding stability* as the similarity in pod assignments between two consecutive executions of the podding algorithm on overlapping object sets. Let $\mathcal{U}_1$ and $\mathcal{U}_2$ be two sets of objects. Given two mappings of podding decisions $A_1 : \mathcal{U}_1 \longrightarrow$ {split, bundle, split-final} and similarly A_2, define podding similarity between A_1 and A_2 as

$$\text{Sim}(A_1, A_2) = \frac{|\{u \in \mathcal{U}_1 \cap \mathcal{U}_2 \mid A_1(u) = A_2(u)\}|}{|\mathcal{U}_1 \cap \mathcal{U}_2|} \quad (6)$$

Under LGA, the podding is *stable*, $\text{Sim}(A_i, A_{i+1}) = 1$. Because LGA memoizes past decisions, it acts as if all objects in $\mathcal{U}_i$ arrive first, followed by all objects in $\mathcal{U}_{i+1} \setminus \mathcal{U}_i$. In this order, LGA—a one-pass deterministic algorithm—reproduces exactly the same decisions for $\mathcal{U}_i \cap \mathcal{U}_{i+1} \subseteq \mathcal{U}_i$ in both runs, yielding $\text{Sim}(A_i^{\text{LGA}}, A_{i+1}^{\text{LGA}}) = 1$.

7.4 Generalizability and Portability

Chipmink 's abstractions and architecture are designed to generalize beyond the Python runtime and port to other language ecosystems. As discussed in §3.3 and §3.4, Chipmink requires two key properties from the host environment: (1) a serialization protocol that can express object structure and shared references, and (2) code execution locality. These requirements are satisfied by many modern runtimes. For example, Java and Rust are natural targets due to their built-in serialization protocols [29, 33], well-defined object identity, and controlled memory models. C++ is also compatible in principle, but requires programmer cooperation—specifically, disciplined use of pointers and the adoption of sufficiently generic serialization. Python is an ideal starting point for Chipmink due to its popularity, persistence needs, extensibility, and generic built-in

Table 1: Real notebooks: five benchmark notebooks for evaluation and three training notebooks for volatility model (§5.2).

Notebook	Topic	Dataset	# Ckpts	# Objects
`skltweet` [96]	Sentiment analysis	185 MB	44	155 K
`ai4code` [46]	EDA codes and comments	4.2 GB	21	20 M
`agripred` [84]	Drought Image classification	6.8 GB	15	214
`msciedaw` [9]	EDA single-cell integration	27 GB	28	424 K
`ecomsmph` [63]	E-commerce data mining	14 GB	41	35 M
`buildats` [88]	Asset trading strategy	303 MB	42	210 K
`storesfg` [14]	Time series forecasting	119 MB	38	90 K
`itsttime` [97]	Sport forecasting	163 MB	54	161 K

serialization (e.g., Pickle), which made it a natural environment to prototype and evaluate our ideas.

7.5 Implementation

We implement Chipmink in Python. Our podding inherits from Dill [66, 67] as the serialization base for its extensive coverage of all objects in our dataset; however, we have also tested Chipmink on Pickle [40] and Cloudpickle [25] with similar results. For this evaluation, we set the thesaurus size to 1 GB (much larger than needed in experiments), c_{pod} to 1200, and `MAX_POD_DEPTH` to 3.

We train a LightGBM [39] as the composable volatility model λ on `buildats`, `storesfg`, and `itsttime` by bootstrapping the podding process to collect the mutation object samples. We obtain over 470k object samples where 7.3% of those are mutating objects.

8 EXPERIMENTS

This section empirically evaluates Chipmink and its components on real notebooks with our reproducibility package published [7]. Our experiment pipeline shows that:

(i) End-to-end Performance. Chipmink offers drastically faster and more compact saving (§§ 8.1 to 8.3) and loading target variables (§8.4). We further analyzes Chipmink performance on parameterized benchmarks §§ 8.5 and 8.6 to reveal its versatility.

(ii) Generalizability and Internals. Chipmink can handle real-world workloads well due to the combination of its internal components. To study their contributions, we perform ablation studies including those for podding (§8.8), podding optimization (§8.7), asynchronous saving (§8.9 and §8.10), and storage layer (§8.11).

Setup. Our testbed is a NUMA machine with 2× AMD EPYC 7552, 1 TB RAM, and about 800 GB of free disk space (SSD) to accommodate `ecomsmph` experiments on inefficient baselines. Nonetheless, all experiments on Chipmink are possible with at least 12 GB of free disk space (not including dataset) and 32 GB of RAM. We run all experiments under a Ubuntu 22.04 Docker container running Python 3.12 without GIL[8] to enable thread-based parallelism.

Dataset. Our dataset consists of eight real computational notebooks (Table 1) and five Python scripts (Table 2). The testing notebook dataset includes `skltweet`, `ai4code`, `agripred`, `msciedaw`, and `ecomsmph`, ranging from hundreds to tens of millions of objects, from exploratory data analysis (EDA) to image classification.. Three notebooks with relatively fewer objects and smaller state sizes are held out as training notebooks for the volatility model: `buildats`,

[7]https://github.com/illinoisdata/pod

[8]https://github.com/colesbury/nogil-3.12-final

Table 2: Real Python scripts in our dataset with their topics, dataset sizes, and numbers of checkpoints and objects.

Script	Topic	Dataset	# Ckpts.	# Objects
`netmnist` [78]	Digit classification	116 MB	43	2.7 K
`rlactcri` [80]	Reinforcement learning	N/A	256	94 K
`vaenet` [79]	Image auto-encoding	116 MB	15	3.1 K
`tseqpred` [81]	Time sequence prediction	24 MB	46	5.2 K
`wordlang` [83]	Language modeling	12.3 MB	83	35 K

`storesfg`, and `itsttime`. On the other hand, we retrieve real Python scripts `netmnist`, `rlactcri`, `vaenet`, `tseqpred`, and `wordlang` from the PyTorch showcasing repository[9] for their completeness, conciseness, and diversity in data science topics and insert checkpoints to save all variables in global and local namespaces.

Baselines. We include several object store baselines to verify the effectiveness of Chipmink and its techniques. *Dill* saves snapshots of the namespace after cell executions serialized by Dill [66, 67]. On partial loading, it loads the entire snapshot and selects only target variables. As a part of the Python standard library, *Shelve* [41] naturally offers persistent namespace through its dictionary interface. To handle loading past states, we save versioned variables as entries `<tid>:<variable_name>` in a single Shelve database. For ZODB [43], two approaches are valid to enable time traveling: *ZODB* and *ZODB-Hist*. In ZODB, we save variables as entries where different versions of variables are stored under separate database paths. Instead of the separation, in ZODB-Hist, we store different versions all under one database path and load past variables through the historical connection feature [42]. Finally, *CRIU* [31] is a middleware checkpoint/restore system. To checkpoint Python namespace, we use CRIU to save a forked Python process having the target namespace and then load the persisted process to extract target variables. We limit the maximum storage usage to 768 GB and terminate early when the system exceeds the limit.

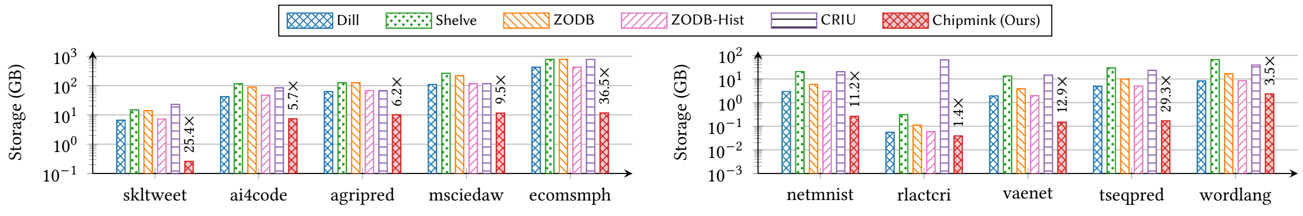
8.1 Chipmink Lowers Storage Requirements

This experiment demonstrates whether Chipmink lowers the storage space required to store namespace states.

More compact storage. Chipmink reduces the storage usages to 5.7–36.5× smaller than the best baseline on notebooks and 1.4–29.3× smaller on scripts (Fig 8a). Most impressively, users can checkpoint `ecomsmph` through Chipmink using only 12 GB of disk space while they need to set aside 426 GB of free disk space when using the best baseline (Dill). A higher storage saving compared to baselines (e.g., in `ecomsmph` and `tseqpred`) indicate that the notebook/script mutates a small part of the object graph, and vice versa (e.g., in `ai4code` and `rlactcri`). On the other hand, Shelve uses the most space across datasets, because of its poor handling of shared references resulting in both incorrect and duplicate data. For example, in `msciedaw`, Shelve breaks the reference from `analyze_multiome_x` to `cell_summary`, resulting in an inconsistent state after loading where `analyze_multiome_x` does not update `cell_summary` as it should.

Effect of mutation rate. We further analyze which types of tasks benefit most from Chipmink by grouping our benchmarks by their estimated mutation rates, i.e., the fraction of objects modified per checkpoint. At very low mutation rates (< 2%), such as in `ecomsmph`

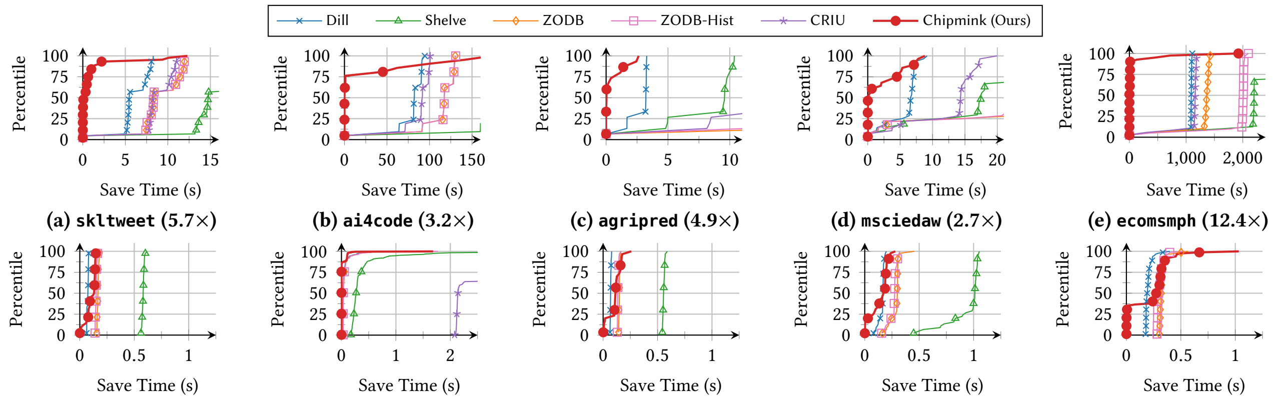
[9]https://github.com/pytorch/examples



(a) Total storage usage (notebooks)

(b) Total storage usage (scripts)

Figure 8: Chipmink stores all variables with 5.7–36.5× smaller storage on notebooks and 1.4–29.3× smaller on scripts than the best baselines. The plots show the total storage required when saving all variables in the namespace at different points in time.



(a) skltweet (5.7×)
(b) ai4code (3.2×)
(c) agripred (4.9×)
(d) msciedaw (2.7×)
(e) ecomsmph (12.4×)
(f) netmnist (0.69×)
(g) rlactcri (3.0×)
(h) vaenet (0.63×)
(i) tseqpred (1.1×)
(j) wordlang (0.93×)

Figure 9: Empirical cumulative distributions (eCDFs) of the perceived saving latency (closer to the top-left corner are better). Numbers in parentheses are Chipmink's speedup over the best baseline; Chipmink stores all variables 2.7–12.4× faster than the best baselines on computational notebooks while remaining competitive with the fastest baselines on scripts.

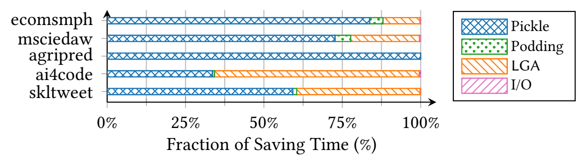
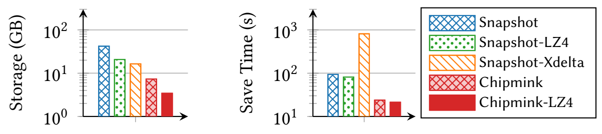


Figure 10: Stepwise breakdown of Chipmink's saving time.



(a) Total storage size

(b) Average saving time

Figure 11: Compression helps but Chipmink remains necessary to efficiently store massive object graphs (e.g., skltweet).

(0.3%), `tseqpred` (1.2%), and `skltweet` (1.7%), only small batches or short-lived tensors change per step. In these cases, Chipmink achieves the largest gains—up to 36.5× over the best baseline—because pods precisely capture fine-grained dispersed updates that existing storages cannot. At low but broader mutation rates (2–10%), including `vaenet` (4.6%), `netmnist` (6.7%, and `msciedaw` (7.3%), model weights and partial dataset scans are updated between checkpoints; here, Chipmink still consistently outperforms baselines by one order of magnitude. For medium mutation rates (10–15%), as in `agripred` (10%), and `ai4code` (13%), more of the graph is touched (e.g., convolutional feature maps or recomputed EDA statistics), reducing relative savings but still showing clear efficiency improvements. Finally, at higher mutation rates ($>$ 15%), such as in `wordlang` (27%) and `rlactcri` (70%), most objects are updated between checkpoints (language model activations, reinforcement learning buffers), where Chipmink 's advantage shrinks but remains non-trivial (1.4–3.5×). These results demonstrate that Chipmink is effective for ML training and data analysis tasks with dispersed, fine-grained updates, while retaining meaningful efficiency even when large portions of the object graph are modified.

8.2 Chipmink Saves Objects Quickly

Next, we test whether Chipmink reduces the perceived saving latency. Here we measure the perceived saving latency by timing all execution delays including the times to synchronously save and acquire locks (if any).

Lower perceived saving latency. eCDF plots (Fig 9) describe (1) proportions of saving latencies above/below a length of time, (2) latency percentiles, and (3) total latency speedups. For example, Chipmink users would not perceive any saving latency beyond 23 seconds (75th percentile think time [102]) on `skltweet`, `agripred`, and `msciedaw`, but would on 7/21 and 4/41 cell executions on `ai4code` and `ecomsmph` respectively. In all notebooks, half of the cell executions are not blocked by Chipmink more than 1 second, as opposed to the median saving latencies between 3 seconds (`agripred`) and 16 minutes (`ecomsmph`) of the best baseline. Measurements on Python scripts reveal similar patterns; however, Chipmink's podding overhead can be observed when the namespace is small (e.g., `netmnist` and `vaenet`) or when most variables are connected (e.g., `wordlang` through its token dictionary).

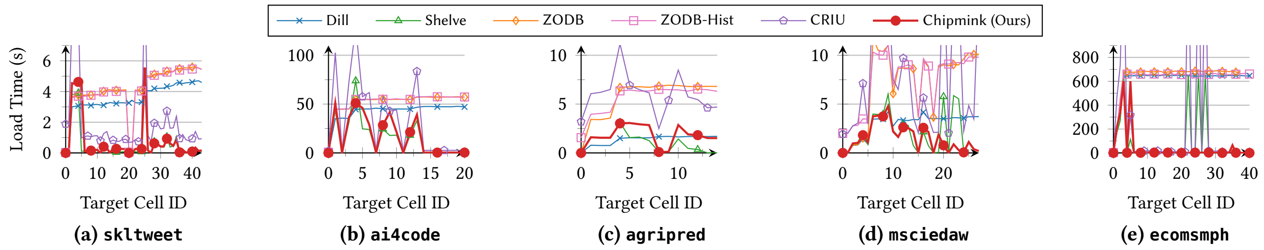


Figure 12: Partial loading time when users are interested in variables accessed at each cell. Chipmink quickly loads target variables proportionally to their sizes, whereas some baselines' performance depends on the entire namespace size.

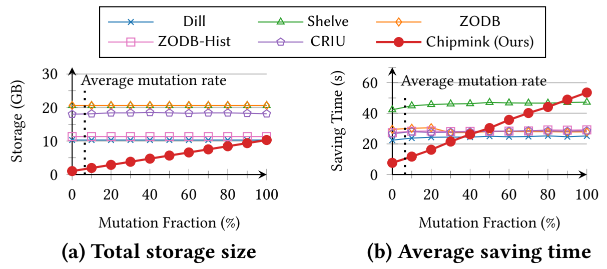


Figure 13: Storage and save time when the notebook mutates 1 GB of data over 10 cells at varied rates. The dotted lines display the mutation fraction averaged over 5 real notebooks.

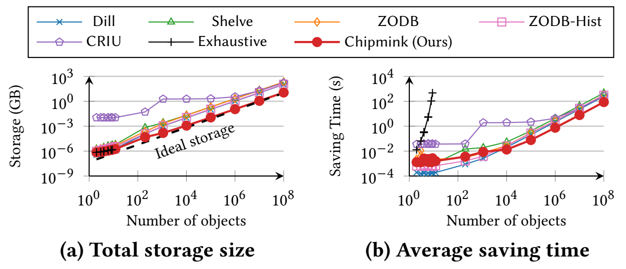


Figure 14: Storage and save time as namespace size scales.

Latency breakdown. Fig 10 shows a stepwise breakdown of Chipmink's time cost. The dominant costs arise from Python Pickle's serialization (e.g., 83.7% in `ecomsmph`, 99.9% in `agripred`) and from LGA (e.g., 65.3% in `ai4code`), where the latter is dominated by evaluating the volatility model λ over millions of objects. Other steps such as I/O and index maintenance are negligible ($\leq$ 5%). These results suggest clear optimization opportunities: a more lightweight volatility model, parallelized object serialization, or more efficient serializers could further reduce latency.

8.3 Chipmink and Byte-Level Compression

While byte-level delta techniques such as LZ4 [105] and Xdelta [64] can help reduce storage size, they do not address the inefficiency of storing massive evolving object graphs. As shown in Fig 11, Chipmink remains necessary to quickly and compactly capture fine-grained object deltas. Nonetheless, Chipmink is fully compatible with compression: applying LZ4 to pod data further reduces storage footprint. In any case, compression's effect on saving time varies depending on its overhead and I/O reduction.

8.4 Chipmink Loads Target Variables Quickly

This experiment tests whether Chipmink can load target variables quickly. After saving all states, we randomly select a target cell ID one by one and request loading the variables accessed in the cell.

Faster loading. By avoiding loading irrelevant variables, Chipmink loads target variables with less delay than Dill, ZODB-Hist, ZODB, and CRIU (Fig 12) whose loading times depend on namespace sizes rather than requested variable sizes. Shelve performs just as fast; however, users should beware of Shelve's broken shared references.

8.5 Chipmink Captures Partial Changes

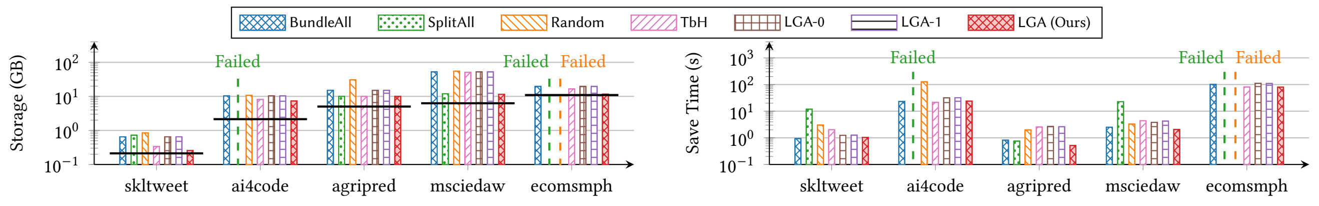
This experiment examines whether Chipmink's storage usage and saving times grow proportionally to object mutations. It synthetically creates a notebook with 100 lists containing 100k byte strings, each of size 100 bytes (total namespace size is roughly 1 GB). Over the next 9 cells, the notebook mutates a fraction of the lists varied from 0% to 100%.

Closely captured object mutations. As expected, Chipmink's storage usage and saving time match with mutation fractions (Fig 13). As objects mutate more from 0% to 100%, Chipmink's storage usage scales from 1 GB (the original namespace size) to 10 GB (all objects always change), converging to snapshotting methods like Dill and ZODB-Hist (Fig 13a) For this namespace structure, Chipmink's saving time is faster than baselines when about 35% of objects mutate and is slower than Dill when more objects mutate (Fig 13b).

8.6 Chipmink Scales with Data Sizes

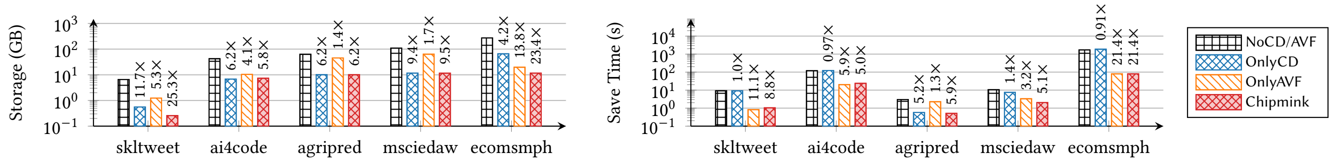
This experiment studies how Chipmink's storage and saving times grow as the number of objects increases. It creates (1) a *small-scale notebook* with 2 cells that randomly mutate {1, 2, 3} lists containing {1, 2, 3} byte strings, each of size 100 bytes, and (2) a *normal-scale notebook* with 10 cells that randomly mutate 1% of 100 lists containing {1, 10, 100, ..., 1M} byte strings, each of size 100 bytes.

Scaling with mutation rate. Chipmink scales both performance dimensions linearly with the number of objects (Fig 14). At a small scale, we measure Chipmink's metadata of around 600 bytes (Fig 14a). As object count increases, Chipmink's storage usage converges to the ideal storage usage as we observe in §8.5. In terms of saving time, Chipmink has a 1.3 ms overhead, making it slower than Dill and ZODB-Hist at small scale (Fig 14b). As object count increases, Chipmink saves quicker than all baselines by leveraging inactive variables. In this synthetic benchmark (1% mutation), Chipmink's throughput is around 1.16 million objects per second.



(a) Total storage usage **(b) Average saving time**

Figure 15: LGA is the most effective podding optimizer in discovering compact podding compared to naive methods (BundleAll, SplitAll, Random), manually derived heuristic (TbH), and LGA with inaccurate volatility models (LGA-0, LGA-1). Thick horizontal lines indicate loose theoretical lower bounds of the optimal storage costs. The Exhaustive baseline is studied in Fig 14.



(a) Total storage size **(b) Average saving time**

Figure 16: Change detector (CD) and active variable filter (AVF) both contribute to storage savings and speedups.

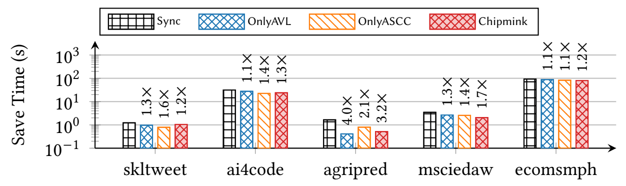


Figure 17: Asynchronous saving reduces saving time further with active variable locking (AVL) and allowlist-based static code checker (ASCC). Chipmink applies both AVL and ASCC.

Optimality at small scale. Compared to the exhaustive search (i.e., trying all 2^{decision} combinations of podding decisions), Chipmink finds > 99.19% optimally compact solution made by the exhaustive search (Fig 14a). Unfortunately, as podding is an NP problem, we can only afford to test the exhaustive search at a small scale; at 18 decisions, Exhaustive takes 1.3 hours to execute (Fig 14b).

8.7 Quality of Internal Podding Optimizer

This experiment tests whether Chipmink's podding optimizer LGA can group objects during podding more compactly and quickly than 6 following alternative podding optimizers. (1) BundleAll bundles all objects. (2) SplitAll splits objects into their pod. (3) Random randomly samples actions uniformly. (4) TbH is a type-based heuristic tuned manually through extensive benchmarks . (5) LGA-0 and (6) LGA-1 refer to the learned greedy algorithm (LGA) with inaccurate volatility models $\lambda_{\text{LGA-0}}(u) = 0$ and $\lambda_{\text{LGA-1}}(u) = 1$. Although the podding problem is an NP problem, we gauge the optimal storage requirement with a loose theoretical lower bound: the maximum namespace size of the notebook (in bytes).

Still more compact storage. LGA consistently produces smaller storage solutions across all notebooks (Fig 15a). On some notebooks, SplitAll and TbH are competitive methods in terms of storage usage. In addition, the gaps between Chipmink and LGA-0/LGA-1 suggest the importance of the volatility model. Compared to the loose lower bounds of storage costs, LGA finds near-optimal podding solutions on `skltweet` and `ecomsmph` while occupying 3.4×, 1.8×, and 2.0× larger storage for `ai4code`, `agripred`, and `msciedaw`.

Table 3: Allowlist-based static code checker (ASCC) varies in its recall and accuracy, but never produces a false negative.

Notebooks	Precision	Recall	Accuracy	#Stable Ser.
`skltweet`	100%	57.1%	86.4%	38
`ai4code`	100%	100%	100%	7
`agripred`	100%	100%	100%	8
`msciedaw`	100%	100%	100%	3
`ecomsmph`	100%	50%	91.7%	11

Still faster saving. Furthermore, LGA makes decisions quickly without significant time overhead (Fig 15b). BundleAll is competitively fast in this regard because it reverts to picking altogether, but it remains slower due to ineffective podding. Conversely, SplitAll and Random are much slower and sometimes time out since they split objects into too many pods for Chipmink to process.

8.8 Impact of CD and AVF

This experiment validates design decisions in §4 where baselines NoCD/AVF, OnlyCD, and OnlyAVF accordingly enable and disable change detector (CD) and active variable filter (AVF). By comparing to NoCD/AVF, OnlyCD and OnlyAVF saves 4.2–11.7× and 1.4–13.8× storage (Fig 16a) with no clear winner (e.g., OnlyCD in `msciedaw`, OnlyAVF in `ecomsmph`), suggesting that both play crucial roles in Chipmink's effectiveness. Change detector helps skipping writing many synonymous pod bytes to storage, while active variable filter removes metadata overheads to manage synonymous pods. On the other hand, active variable filter (i.e., OnlyAVF and Chipmink) effectively bypasses object serializations, and thus, is crucial for the reduction in overhead time 2.2–11.1× faster compared to NoCD/AVF. In this aspect, the change detector can only cut down the I/O time, smaller compared the time to serialize the massive number of objects in most notebooks, except `agripred`.

8.9 Impact of Asynchronous Saving

To study the effectiveness of asynchronous saving techniques, we implement Chipmink without asynchronous saving Sync, and two ablation baselines OnlyAVL and OnlyASCC as Chipmink with and without active variable locking (AVL) and allowlist-based static

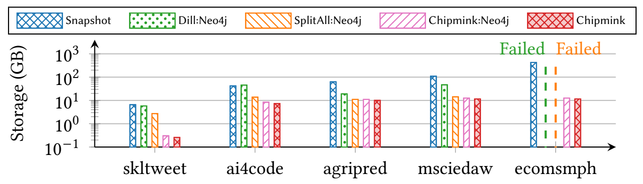


Figure 18: Storage usage of graph object stores under different configurations: without delta identification (Dill:Neo4j), with naïve delta identification that splits all objects into separate pods (SplitAll:Neo4j), and with our LGA (Chipmink:Neo4j).

code checker (ASCC) described in §6. Despite the quick succession of cell executions in our benchmark, both asynchronous saving via active variable locking (AVL) and allowlist-based static code checker (ASCC) are required to improve Chipmink's distribution of saving times over synchronous saving Sync (Fig 17).

8.10 ASCC Accuracy

We evaluate the allowlist-based static code checker (ASCC) across our benchmark notebooks by confirming the equality in serialized bytes (Table 3), only in cases when pickled bytes are deterministic (noted as numbers of stable serialization). ASCC achieves 100% precision in all cases, meaning it never produces false positives; this correctness guarantee ensures that any code cells it flags as static are indeed safe to execute with an ongoing saving. However, recall varies across workloads, ranging from 50–100%. For example, in `skltweet` ASCC only identifies 57.1%, leaving 42.9% unrecognized for potential optimization. Overall, ASCC is conservative by design: it guarantees correctness by avoiding false positives, while false negatives highlight directions for extending the checker to better capture stability patterns in complex notebooks.

8.11 Comparison with Graph Storage

This experiment highlights the importance of Chipmink for storage efficiency in graph object stores. Without delta identification, Dill:Neo4j consumes substantially more space, redundantly writing large portions of the object graph. Chipmink remains necessary to significantly reduce storage over the naïve strategy (SplitAll:Neo4j) by grouping co-mutating objects. The choice of backend store makes only a small difference in space: Chipmink:Neo4j uses at most 17% more storage than Chipmink (LGA with a specialized key–value store). Saving latency is a bigger differentiator: Neo4j is currently slower by up to 104×, 59×, and 58× for Dill:Neo4j, SplitAll:Neo4j, and Chipmink:Neo4j respectively, compared to Dill, due to the cost of updating millions of objects or pods as individual nodes at each save. We expect this gap can be narrowed by improving Neo4j's batch write interface and by implementing write-optimized data structures.

9 RELATED WORKS

This section lists and discusses related works to Chipmink in the context of object storage and persistence for computational notebooks, temporal exploration, and podding optimization.

Python object storages. Only a few Python object stores correctly persist interdependent objects but none implement a partial object store. ZODB [43, 57] is a mature object storage that correctly handles object references [106] with many database features like historical connection and concurrent transactions. Meanwhile, other object storages only support simplified object types. For example, Shelve [41], redis_shelve[52], and Chest [65] (no longer maintained) do not support shared references across entries. pySOS [11] and pickleDB [34] (despite its name) only support JSON-able objects. Nonetheless, to the best of our knowledge, all of the existing works store snapshots of the objects in their entirety; Chipmink is the first Python object store that partially saves and loads objects, making temporal exploration feasible.

Object databases. Object-oriented database management systems (OODBMS) (e.g., [8, 17, 27, 50, 56, 72, 74]) may seem like natural solutions for an object store. However, these OODBMS'es mainly focus on relaxing the first normal form (1NF) [26] to manage user-defined complex structures, rather than persisting general execution states with rapid mutations addressed by Chipmink. As a result, relying on an OODBMS may restrict users to a few object types, require manual work, and incur excessive storage overheads.

Graph databases. Graph databases like Neo4j [4] and Dgraph [1] primarily support storage of the current graph state, offering only coarse-grained versioning through transaction logs or application-level timestamping [10]. Object–graph layers such as Neomodel and Renesca for Neo4j [5, 6] and GQLAlchemy for Memgraph [2] provide higher-level APIs but do not change the underlying storage semantics, and thus inherit the same limitations in fine-grained delta tracking. Similarly, newer systems such as Kùzu Graph Store [3] are designed for efficient graph queries but still operate at vertex/edge granularity. Temporal graph systems extend these designs by recording historical states using either model-based encodings (timestamped nodes/edges) or snapshot-based checkpoints [18–20, 47], where their coarse-grained storage would incur significant overheads at the scale of fine-grained evolving object graphs. In contrast, Chipmink proposes a *delta identification* over evolving in-memory (i.e., out-of-storage) graphs : its podding algorithm pinpoints fine-grained changes across millions of objects , achieving a level of efficiency and precision that coarse-grained existing graph database techniques cannot provide.

10 CONCLUSION

Through our novel idea of subgraph deltas, Chipmink is an object store with an efficient delta identification for massive and evolving object graphs. Unlike existing tools, Chipmink accurately, comprehensively, and efficiently identifies deltas, achieved through several new techniques such as podding, virtual memo space, change detector, synonym resolver, active variable filter, learned greedy algorithm for podding optimization, asynchronous saving, active variable locking, and allowlist-based static code checker. Our empirical experiments show Chipmink is faster and requires less storage space than existing solutions and alternative designs.

11 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Awards #2440498 and #2312561, and by the National Center for Supercomputing Applications.

REFERENCES

[1] [n.d.]. Dgraph. https://dgraph.io/. Accessed: 2025-08-18.

[2] [n.d.]. GQLAlchemy. https://github.com/memgraph/gqlalchemy. Accessed: 2025-08-18.

[3] [n.d.]. Kùzu: Embedded, scalable, blazing fast graph database. https://kuzudb.com/. Accessed: 2025-08-18.

[4] [n.d.]. Neo4j: Graph Database & Analytics. https://neo4j.com/. Accessed: 2025-08-18.

[5] [n.d.]. Neomodel: An Object Graph Mapper (OGM) for the Neo4j graph database. https://github.com/neo4j-contrib/neomodel. Accessed: 2025-08-18.

[6] [n.d.]. Renesca: Scala library for the Neo4j REST API. https://github.com/renesca/renesca. Accessed: 2025-08-18.

[7] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming Graph Partitioning: An Experimental Study. *Proc. VLDB Endow.* 11, 11 (2018), 1590–1603. https://doi.org/10.14778/3236187.3236208

[8] Actian. [n.d.]. NoSQL object databases. https://www.actian.com/databases/nosql/. [Online; accessed May-2-2024].

[9] AmbrosM. 2022. MSCI EDA which makes sense. https://www.kaggle.com/code/ambrosm/msci-eda-which-makes-sense. [Online; accessed September-8-2024].

[10] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L Reutter, and Domagoj Vrgoc. 2016. Foundations of modern graph query languages. *CoRR, abs/1610.06264* (2016).

[11] Lawrie Brown Arnaud Dagnelies. [n.d.]. pySOS: Simple Objects Storage. https://github.com/dagnelies/pysos. [Online; accessed May-2-2024].

[12] Ben Baumer, Mine Cetinkaya-Rundel, Andrew Bray, Linda Loi, and Nicholas J. Horton. 2014. R Markdown: Integrating A Reproducible Analysis Tool into Introductory Statistics. *Technology Innovations in Statistics Education* 8, 1 (2014). https://doi.org/10.5070/t581020118

[13] Benjamin Baumer and Dana Udwin. 2015. R markdown. *Wiley Interdisciplinary Reviews: Computational Statistics* 7, 3 (2015), 167–177.

[14] Ekrem Bayar. 2022. Store Sales TS Forecasting - A Comprehensive Guide. https://www.kaggle.com/code/ekrembayar/store-sales-ts-forecasting-a-comprehensive-guide/notebook. [Online; accessed April-26-2024].

[15] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *Comput. Surveys* 56, 2 (2023), 1–40.

[16] Niv Buchbinder, Moran Feldman, Joseph Seffi, and Roy Schwartz. 2015. A tight linear time (1/2)-approximation for unconstrained submodular maximization. *SIAM J. Comput.* 44, 5 (2015), 1384–1402.

[17] Paul Butterworth, Allen Otis, and Jacob Stein. 1991. The Gemstone Object Database Management System. *Commun. ACM* 34, 10 (1991), 64–77. https://doi.org/10.1145/125223.125254

[18] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2019. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering* 32, 3 (2019), 424–437.

[19] Borui Cai, Yong Xiang, Longxiang Gao, He Zhang, Yunfeng Li, and Jianxin Li. 2023. Temporal Knowledge Graph Completion: A Survey. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 6545–6553. https://doi.org/10.24963/IJCAI.2023/734

[20] Alexander Campos, Jorge Mozzino, and Alejandro Vaisman. 2016. Towards temporal graph databases. *arXiv preprint arXiv:1604.08568* (2016).

[21] 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 *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–12. https://doi.org/10.1145/3313831.3376729

[22] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[23] 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.

[24] R. D. Clarke. 1946. An application of the Poisson distribution. *Journal of the Institute of Actuaries* 72, 3 (1946), 481–481. https://doi.org/10.1017/S0020268100035435

[25] cloudpickle. [n.d.]. cloudpickle: Extended pickling support for Python objects. https://github.com/cloudpipe/cloudpickle. [Online; accessed May-3-2024].

[26] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[27] George P. Copeland and David Maier. 1984. Making Smalltalk a Database System. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 316–325. https://doi.org/10.1145/602259.602300

[28] Microsoft Corporation. [n.d.]. lightgbm.Boost: feature_importance. https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.Booster.html#lightgbm.Booster.feature_importance. [Online; accessed May-3-2024].

[29] Oracle Corporation. [n.d.]. Serializable (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html. [Online; accessed August-6-2025].

[30] cppreference.com. [n.d.]. cv (const and volatile) type qualifiers. https://en.cppreference.com/w/cpp/language/cv. [Online; accessed May-3-2024].

[31] CRIU. [n.d.]. CRIU: Main page. https://criu.org/Main_Page. [Online; accessed September-19-2024].

[32] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–34.

[33] Serde Developers. [n.d.]. Serde: Serialization framework for Rust. https://serde.rs/. [Online; accessed August-6-2025].

[34] Harrison Erd. [n.d.]. pickleDB - simple key-value database. https://pythonhosted.org/pickleDB/. [Online; accessed May-2-2024].

[35] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+ Data Space Versioning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–17.

[36] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Large-scale Evaluation of Notebook Checkpointing with AI Agents. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–8.

[37] Uriel Feige, Vahab S. Mirrokni, and Jan Vondrák. 2011. Maximizing Non-monotone Submodular Functions. *SIAM J. Comput.* 40, 4 (2011), 1133–1153. https://doi.org/10.1137/090779346 arXiv:https://doi.org/10.1137/090779346

[38] Python Software Foundation. [n.d.]. Command line and environment. https://docs.python.org/3/using/cmdline.html. (Date accessed: January 16 2024).

[39] Python Software Foundation. [n.d.]. LightGBM Python Package. https://pypi.org/project/lightgbm. [Online; accessed September-30-2024].

[40] Python Software Foundation. [n.d.]. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html. [Online; accessed May-3-2024].

[41] Python Software Foundation. [n.d.]. shelve — Python object persistence. https://docs.python.org/3/library/shelve.html. [Online; accessed May-2-2024].

[42] Zope Foundation. [n.d.]. Historical Connections. https://zodb.org/en/latest/historical_connections.html. [Online; accessed May-3-2024].

[43] Zope Foundation. [n.d.]. ZODB - a native object database for Python. https://zodb.org/en/latest/. [Online; accessed May-2-2024].

[44] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and applications* 13 (2010), 113–129.

[45] Sam Gross. 2023. *Making the Global Interpreter Lock Optional in CPython.* PEP 703. https://peps.python.org/pep-0703/

[46] Sanskar Hasija. 2022. AI4Code Detailed EDA. https://www.kaggle.com/code/odins0n/ai4code-detailed-eda. [Online; accessed April-26-2024].

[47] Jiamin Hou, Zhanhao Zhao, Zhouyu Wang, Wei Lu, Guodong Jin, Dong Wen, and Xiaoyong Du. 2024. AeonG: An Efficient Built-in Temporal Support in Graph Databases. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1515–1527. https://doi.org/10.14778/3648160.3648187

[48] Shengya Huang, Zhaoheng Li, Derek Werner, and Yongjoo Park. 2025. MojoFrame: Dataframe Library in Mojo Language. *arXiv preprint arXiv:2505.04080* (2025).

[49] James J Hunt, Kiem-Phong Vo, and Walter F Tichy. 1998. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 192–214.

[50] InterSystems. [n.d.]. InterSystems IRIS: Database Management. https://www.intersystems.com/data-platform/database-management/. [Online; accessed May-2-2024].

[51] Ipoom Jeong, Jinghan Huang, Chuxuan Hu, Dohyun Park, Jaeyoung Kang, Nam Sung Kim, and Yongjoo Park. 2025. UPP: Universal Predicate Pushdown to Smart Storage. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 419–433.

[52] Niels van Huijstee Jose Tiago Macara Coutinho. [n.d.]. Redis Shelve. https://github.com/FugaCloud/redis-shelve. [Online; accessed May-2-2024].

[53] Project Jupyter. [n.d.]. JupyterLab. https://github.com/dagnelies/pysos. [Online; accessed May-2-2024].

[54] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.

[55] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. https://eprints.soton.ac.uk/403913/

[56] Charles Lamb, Gordon Landis, Jack A. Orenstein, and Daniel Weinreb. 1991. The ObjectStore Database System. *Commun. ACM* 34, 10 (1991), 50–63. https://doi.org/10.1145/125223.125244

[57] Reuven M Lerner. 2002. At the forge: Databases and Zope. *Linux Journal* 2002, 97 (2002), 9.

[58] Zhaoheng Li, Supawit Chockchowwat, Hanxi Fang, and Yongjoo Park. 2025. Demo of Kishu: Time-Traveling for Computational Notebooks. In *Companion of the 2025 International Conference on Management of Data*. 167–170.

[59] 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.

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

[61] 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

[62] Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. Sieve: Effective filtered vector search with collection of indexes. *arXiv preprint arXiv:2507.11907* (2025).

[63] Lucas HK Liu. 2022. E-Comm Data Mining for Smartphone Products. https://www.kaggle.com/code/lucashkliu/e-comm-data-mining-for-smartphone-products. [Online; accessed September-8-2024].

[64] Josh MacDonald. 2000. *File system support for delta compression.* Ph.D. Dissertation. Masters thesis. Department of Electrical Engineering and Computer Science ....

[65] Max Hutchinson Matthew Rocklin. [n.d.]. Chest: A dictionary that spills to disk. https://github.com/mrocklin/chest. [Online; accessed May-2-2024].

[66] M McKerns and M Aivazis. 2010. Pathos: a framework for heterogeneous computing. *See http://trac. mystic. cacr. caltech. edu/project/pathos* (2010).

[67] Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. 2011. Building a Framework for Predictive Science. In *Proceedings of the 10th Python in Science Conference 2011 (SciPy 2011), Austin, Texas, July 11 - 16, 2011*, Stéfan van der Walt and Jarrod Millman (Eds.). scipy.org, 76–86. https://doi.org/10.25080/MAJORA-EBAA42B7-00D

[68] 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.

[69] MySQL. 2024. MySQL 8.4 Reference Manual: 17.5.1 Buffer Pool. https://dev.mysql.com/doc/refman/8.4/en/innodb-buffer-pool.html. (Date accessed: July 10 2024).

[70] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. 2006. Fast suboptimal algorithms for the computation of graph edit distance. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 163–172.

[71] NumPy. [n.d.]. Routines: Input and output. https://numpy.org/doc/stable/reference/routines.io.html. [Online; accessed May-3-2024].

[72] Jack A. Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. 1992. Query Processing in the ObjectStore Database System. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, Michael Stonebraker (Ed.). ACM Press, 403–412. https://doi.org/10.1145/130283.130343

[73] pandas. [n.d.]. User Guide: IO tools (text, CSV, HDF5, ...). https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html. [Online; accessed May-3-2024].

[74] James Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning. 2006. *The Definitive Guide to db4o*. https://doi.org/10.1007/978-1-4302-0176-2

[75] Fernando Pérez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Comput. Sci. Eng.* 9, 3 (2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[76] Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. 2023. Computing graph edit distance via neural graph matching. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1817–1829.

[77] Saurabh Pujari. [n.d.]. object-tracker. https://github.com/saurabh0719/object-tracker. [Online; accessed May-29-2025].

[78] PyTorch. [n.d.]. Basic MNIST Example. https://github.com/pytorch/examples/tree/main/mnist. [Online; accessed January-7-2024].

[79] PyTorch. [n.d.]. Basic VAE Example. https://github.com/pytorch/examples/tree/main/vae. [Online; accessed January-7-2024].

[80] PyTorch. [n.d.]. Reinforcement learning training example. https://github.com/pytorch/examples/tree/main/reinforcement_learning. [Online; accessed January-7-2024].

[81] PyTorch. [n.d.]. Time Sequence Prediction. https://github.com/pytorch/examples/tree/main/time_sequence_prediction. [Online; accessed January-7-2024].

[82] PyTorch. [n.d.]. Tutorials: Saving and Loading Models. https://pytorch.org/tutorials/beginner/saving_loading_models.html. [Online; accessed May-3-2024].

[83] PyTorch. [n.d.]. Word-level Language Modeling using RNN and Transformer. https://github.com/pytorch/examples/tree/main/word_language_model. [Online; accessed January-7-2024].

[84] DS Rahul. 2020. Agricultural Drought Prediction. https://www.kaggle.com/code/dsrhul/agricultural-drought-prediction. [Online; accessed September-8-2024].

[85] Rust. [n.d.]. Rust By Example: Scoping rules. https://doc.rust-lang.org/rust-by-example/scope.html. [Online; accessed May-3-2024].

[86] Rust. [n.d.]. The Rust Programming Language: Understanding Ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. [Online; accessed May-3-2024].

[87] Raunak Shah, Zhaoheng Li, and Yongjoo Park. 2025. QStore: Quantization-Aware Compressed Model Storage. *arXiv preprint arXiv:2505.04081* (2025).

[88] Andrey Shtrauss. 2022. Building an Asset Trading Strategy. https://www.kaggle.com/code/shtrausslearning/building-an-asset-trading-strategy/notebook. [Online; accessed April-26-2024].

[89] Dimitre Trendafilov Nasir Memon Torsten Suel. 2002. zdelta: An efficient delta compression tool. (2002).

[90] Torsten Suel. 2019. Delta compression techniques. *Encyclopedia of Big Data Technologies* 63 (2019).

[91] Hironobu Suzuki. 2024. The Internals of PostgreSQL - 8.Buffer Manager. https://www.interdb.jp/pg/pgsql08.html. (Date accessed: July 10 2024).

[92] Alexander K Taylor, Yicong Huang, Junheng Hao, Xinyuan Lin, Xiusi Chen, Wei Wang, and Chen Li. 2024. Data Science Tasks Implemented with Scripts versus GUI-Based Workflows: The Good, the Bad, and the Ugly. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 267–277.

[93] R Development Core Team. [n.d.]. The R Manuals. https://cran.r-project.org/manuals.html. (Date accessed: January 16 2024).

[94] TensorFlow. [n.d.]. ML basics with Keras: Save and load models. https://www.tensorflow.org/tutorials/keras/save_and_load. [Online; accessed May-3-2024].

[95] Walter F Tichy. 1984. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 309–321.

[96] Cornell University. 2021. SKLearn Tweet Classification. https://github.com/CornellCAC/CVW_PyDataSci2/blob/master/code/sklearn_tweet_classification.ipynb. [Online; accessed April-26-2024].

[97] Theo Viel. 2023. It's That Time of the Year Again. https://www.kaggle.com/code/theoviel/it-s-that-time-of-the-year-again. [Online; accessed April-26-2024].

[98] Chan Kha Vu and Peter Mortensen. [n.d.]. How can I prevent Google Colab from disconnecting? https://stackoverflow.com/questions/57113226/how-can-i-prevent-google-colab-from-disconnecting. (Date accessed: January 16 2024).

[99] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation* 79 (2014), 258–272.

[100] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. 2015. Edelta: A {Word-Enlarging} Based Fast Delta Compression Approach. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*.

[101] Yihui Xie, Joseph J Allaire, and Garrett Grolemund. 2018. *R markdown: The definitive guide.* Chapman and Hall/CRC.

[102] 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. *IEEE Data Eng. Bull.* 44, 1 (2021), 66–78. http://sites.computer.org/debull/A21mar/p66.pdf

[103] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. 2016. A short survey of recent advances in graph matching. In *Proceedings of the 2016 ACM on international conference on multimedia retrieval*. 167–174.

[104] Kaisheng Zeng, Chengjiang Li, Lei Hou, Juanzi Li, and Ling Feng. 2021. A comprehensive survey of entity alignment for knowledge graphs. *AI Open* 2 (2021), 1–13.

[105] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[106] ZODB. [n.d.]. Support for ZODB object serialization. https://github.com/zopefoundation/ZODB/blob/master/src/ZODB/serialize.py. [Online; accessed May-2-2024].

Received 1 June 2025; revised 1 October 2025; accepted 15 November 2025