Runtime Provenance Refinement for Notebooks

Nachiket Deo
University of Connecticut
nachiket.deo@uconn.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Oliver Kennedy
University at Buffalo
okennedy@buffalo.edu

ABSTRACT

Computational notebooks (e.g., Jupyter or Apache Zeppelin) have become a popular choice for data exploration, preparation, and ETL. Notebooks are more suited for interactive development of data pipelines than classical workflow systems, because they provide immediate feedback for the results of a computation and do not require the full computation to be specified upfront. However, the notebook model suffers from poor reproducibility, does not support automatic incremental re-evaluation of code when the code or inputs change, and does not allow for parallel execution of cells — all symptoms of its kernel-based evaluation strategy. We propose a new *"workbook"* model that combines the usability of notebooks with the provenance and parallel execution capabilities of workflow systems. This is made possible through a novel approach that refines a static approximation of provenance for Python code at runtime and a scheduler that dynamically adapts the execution order of cells based on data dependencies detected or refuted at runtime. We demonstrate the feasibility of this approach using a prototype implementation in our notebook engine Vizier.

CCS CONCEPTS

• **Software and its engineering** → *Runtime environments*; • **Information systems** → **Data provenance**;

1 INTRODUCTION

Workflow systems [3] break complex tasks like ETL, model-fitting, and more into a series of smaller, parallelizable steps, but require users to explicitly declare inter-step data dependencies. Computational notebooks like Jupyter instead model tasks as sequences of code 'cells' that each describe a step of the computation without explicit inter-cell dependencies. Users manually trigger cell execution, dispatching the cell's code to a long-lived Python interpreter called a kernel. Cells share data through the state of the interpreter, e.g., a global variable created in one cell can be read by another cell.

Companies like Netflix[1] are increasingly turning to notebooks for bulk ETL and ML workloads as they allow a faster, more interactive development cycle. However, the kernel's hidden state often creates bugs that are hard to understand or reproduce [1, 10][2]. Furthermore, without explicit dependencies, the notebook execution model also precludes inter-cell parallelism and incremental evaluation.

[1]https://netflixtechblog.com/scheduling-notebooks-348e6c14cfd6
[2]https://www.youtube.com/watch?v=7jiPeIFXb6U

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
TaPP'22, June 17, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9349-2/22/06.
https://doi.org/10.1145/3530800.3534535

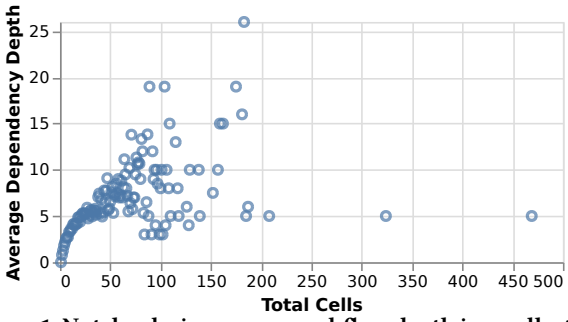


Figure 1: Notebook size versus workflow depth in a collection of notebooks scraped from github [10].

Dynamically collected provenance [9] can address the hidden state problem, but is of limited use for scheduling since dependencies are only learned after running a cell: By the time we learn that two cells can be safely executed concurrently, they are finished. Static dataflow analysis [7] addresses both needs, but requries approximating dependencies — conservatively, limiting opportunities for parallelism; or optimistically, risking errors from missed dependencies. We bridge the static/dynamic gap by proposing a hybrid approach: *Incremental Runtime Provenance Refinement*, where statically approximated provenance is incrementally refined at runtime.

We demonstrate how provenance refinement enables (i) parallel execution and (ii) partial re-evaluation of notebooks. As we show, this requires a fundamental shift away from a single monolithic kernel towards isolated per-cell interpreters, a model we call Isolated Code Execution (ICE). We validate our ideas by extending an ICE notebook named *Vizier* [1]. Cells in Vizier run in isolated interpreters and communicate only through dataflow. We outline the challenges of extending Vizier to support implicit dependencies through incrementally refined provenance. This, in turn allows for parallel and incremental notebook execution, while retaining the typical programming model of Jupyter, where developers do not have to explicitly declare what state to share among cells.

As a preliminary assessment of the potential of parallelizing Jupyter notebooks, we conducted a survey on notebooks scraped from Github by Pimentel et. al. [10]. We only include notebooks using Python and which are known to execute successfully (~6000 notebooks). We constructed a dataflow graph for each notebook as described in Section 5. As a proxy measure for potential speedup, we considered the depth of this graph. Figure 1 presents the depth — the maximum number of cells that must be executed serially — in relation to the total number of Python cells in the notebook. The average notebook has over 16 cells, but an average dependency depth of just under 4 and an average parallelism factor of 4.

We outline our central contribution: **Incremental Runtime Provenance Refinement** in Section 2, and review Vizier's ICE architecture in Section 3. Afterwards, we discuss our remaining contributions: (i) **An Incremental Provenance-based Scheduler**.

```python
import urllib.request as r
with r.urlopen('http://someweb/code.py') as response:
  eval( response.read() )
```

(a) Dynamic code evaluation in Python may lead to arbitrary dependencies that will only be known at runtime.

```python
b = d * 2 if a > 10 else e * 2
```

(b) Static dataflow analysis has to conservatively over-approximate dataflow because control flow depends on the program's input.

Figure 2: Example Python code

In Section 4, we present a scheduler for incremental and parallel notebook execution. We discuss the challenges that arise due to provenance mispredictions and how to compensate for them. (ii) **Jupyter Import**: Section 5 discusses how we extract approximate provenance from Python code statically, and how existing notebooks written for Jupyter can be translated to ICE notebook architectures like Vizier. (iii) **Implementation in Vizier and Experiments**: We have implemented a preliminary prototype of the proposed scheduler in Vizier. Section 6 presents our experiments on the parallel evaluation of Jupyter notebooks.

2 RUNTIME PROVENANCE REFINEMENT

Conservative static analysis for Python either produces a coarse-grained over-approximation of the real data dependencies of a program, or has to allow for missed dependencies. To see why this is the case, consider the code snippet in Figure 2a, which executes a piece of Python code retrieved from the web. The dynamically evaluated code can create data dependencies between everything in the global scope. A further overhead of conservative static analysis is the need to recursively descend into libraries for completeness.

Overly conservative static analysis must accept excessive runtimes to fully analyze all dependent libraries and approximations to manage dynamically executed code, or must instead treat all cells as interdependent. However, a less conservative approach could lead to unsafe notebook execution if it misses a dependency. To overcome this dilemma, we propose an approach that computes approximate provenance using static analysis (allowing for both false negative and false positives in terms of data dependencies) and deals with missing and spurious data dependencies by discovering them and compensating for them at runtime. This approach is sensible in the context of computational notebooks, and prior systems like Nodebook, a Jupyter plugin developed at Stitchfix [12], make similar assumptions.

Static Approximate Provenance. An initial pass over the notebook's code obtains a set of read and write dependencies for each cell using Python's AST library and standard dataflow equations [7] to derive an approximate dataflow graph. To minimize performance overhead, this step only analyzes the user's code and does not consider other modules (libraries) — intra-module dependencies (e.g., stateful libraries) will be missed at this stage, but can still be discovered at runtime. Like any static dataflow analysis, this stage may also produce false positives due to control-flow decisions that depend on the input. For example, in Figure 2b whether the cell has a read dependency on `d` or `e` depends on the *runtime* value of `a`.

Exact Runtime Provenance. As the notebook executes, provenance refinement relies on the ICE architecture to collect data artifacts written to or read by each cell. The resulting dynamically collected read / write sets are used to refine the dataflow graph created by static analysis. Our scheduler (Section 4), assessing opportunities for parallelism or work re-use across notebook re-executions, leverages this refined information as it becomes available.

3 ISOLATED CELL EXECUTION

An isolated cell execution notebook (ICE) isolates cells by executing each in a fresh kernel. Before discussing how notebooks written for monolithic kernels, like Jupyter, can be mapped to the ICE model in Section 5, we first review the key differences between the monolithic approach and systems like Vizier [1] or Nodebook [12].

Communication Model. As in a monolithic kernel notebook, an ICE notebook maintains a shared global state that is manipulated by each individual cell. However, these manipulations are explicit: for a variable defined in one cell (the writer) to be used in a subsequent cell (the reader): (i) the writer must explicitly export the variable into the global state, and (ii) the reader must explicitly import the variable from the global state. For example, Vizier provides explicit setter and getter functions (respectively) on a global state variable, while Nodebook inspects the Python interpreter's global scope dictionary in between cell executions. As mentioned in the introduction, with our scheduler it will no longer be necessary for the user to explicitly call setters and getters.

State Serialization. When a state variable is exported, it is serialized by the Python interpreter and exported into a versioned state management system. We refer to the serialized state as an *artifact*. Each cell executes in the context of a scope, a mapping from variable names to artifacts that can be imported by the cell.

By default, Vizier serializes state through Python's native `pickle` library, but can be easily extended with more specialized codecs like (i) Python code (e.g., function or class definitions) is exported as raw Python code and imported with `eval`. (ii) Pandas dataframes exported in parquet format and hosted through Apache Arrow.

4 SCHEDULER

The semantics of a workbook notebook is the serial execution of the cells in notebook order. We refer to the set of variables imported or exported by each cell as the cell's read and write sets, respectively. A *correct* execution is thus defined in terms of view serializability [11]: A (parallel) schedule is correct iff the artifact versions that are read by each cell are consistent with the versions the cell would read in a serial execution. Note that blind writes are not an issue in Vizier, because writes to an artifact create a new (immutable) version. Thus, cells that blindly write an artifact do not conflict with each other. We assume that cell execution is atomic and idempotent: we are allowed to freely interrupt or restart a cell's execution.

Naive Scheduling. Let N denote a notebook, a sequence of cells $[c_1, \ldots, c_n]$. Assume, initially, that for each cell $c_i \in N$ we are given exact read and write sets ($\mathcal{R}(c_i)$ and $\mathcal{W}(c_i)$ respectively). A notebook's data dependency graph $G = (N, D)$ connects cells through edges $(r, w, \ell) \in D$ labelled with symbols as follows:

$$D = \{\ (c_r, c_w, \ell) \mid c_r, c_w \in N, \ell \in \mathcal{R}(c_r), \ell \in \mathcal{W}(c_w), w < r, \nexists c_{w'} \in N \text{ s.t. } w < w' < r, \ell \in \mathcal{W}(c_{w'})\ \}$$

An edge labelled ℓ exists from any cell c_r that reads symbol ℓ to the most recent preceding cell that writes symbol ℓ.

Denote by $\mathcal{S}(c) \in \{\text{PENDING}, \text{DONE}\}$ the state of a cell (i.e., DONE after it has completed execution); a cell c can be scheduled for execution when all cells connected to incoming edges are DONE: $\forall (c, c_w, \ell) \in D : \mathcal{S}(c_w) = \text{DONE}$. When a cell c_r imports variable ℓ from the global scope, where $(c_r, c_w, \ell) \in D$, it receives the version exported by cell c_w. Any execution order that complies with this rule produces schedules that are view-equivalent to the notebook order and, thus, will produce the same result as a serial execution.

Runtime Refinement. Recall that our static analysis approach produces a dependency graph $\widetilde{G} = (N, \widetilde{D})$ which may have spurious edges and may miss edges. We refine $\widetilde{G}$ at runtime. There are four possible types of changes to the dependency graph when a cell c is executed. In the following we discuss these cases and how to compensate for them to ensure scheduler *correctness*.

(i) When a read does not materialize during c's execution, we remove the corresponding edge from the dependency graph. Such spurious reads of a variable l may cause a delay in c's execution, because c has to wait for the cell writing l to finish execution. However, the correctness of the schedule is not affected. **(ii)** A write of l that does not materialize causes inbound edges with the corresponding label to be redirected to the preceding cell to write l. Cells dependent on c's version of l could not have started yet, so the schedule is still valid. **(iii)** A missed read that materializes during c's execution adds a new edge to the dependency graph. If the edge leads to a cell c' in the PENDING state, the read operation may block until the writing cell has completed. This state is less desirable, as we may have already allocated resources for the blocked cell c which may lead to resource starvation. **(iv)** A missed write of variable l redirects a subset of edges with the corresponding label l to the cell c. This is only a correctness error if one of the dependent cells has already been started — if so, the cell must be aborted and rescheduled after the current cell c completes.

PROPOSITION 4.1 (TERMINATION AND CORRECTNESS). *For any notebook N and approximated dependency graph $\widetilde{G}$ for N, the execution of N using the naive approach with refinement and compensation is guaranteed to terminate and produces a correct schedule.*

Incremental Re-execution. Vizier automatically refreshes dependent cells when a cell c is modified by the user using incremental re-execution which avoids re-execution of cells whose output will be the same in the modified notebook. For that, the modified cell c is put into PENDING state. Furthermore, all cells that depend on c directly or indirectly are also put into PENDING state. That is, we memorize a cell's actual dependencies from the previous execution and initially assume that the dependency graph will be the same as in the previous execution. The exception is the modified cell for which we statically approximate provenance from scratch. During the execution of the modified cell or one of its dependencies we may observe changes to the read and write set of a cell. We compensate for that using the repair actions described above.

5 JUPYTER IMPORT

We now outline the conversion of (monolithic-kernel) Jupyter notebooks into ICE-compatible form. For this preliminary work, we make a simplifying assumption that all inter-cell communication occurs through the kernel's global scope (e.g., as opposed to files).

```
1 def foo(): print(a)
2 a = 1
3 foo()  # Prints '1'
4 def bar():
5   a = 2
6   foo()
7 bar()  # Prints '1'
```

```
1 def foo():
2   def bar(): print(a)
3   a = 2
4   return bar
5 bar = foo()
6 bar()  # Prints '2'
7 a = 1
8 bar()  # Prints '2'
```

Figure 3: Scope capture in Python happens at function definition, but captured scopes remain mutable.

Python's `ast` module provides a structured representation of the code: an *abstract syntax tree* (AST). Variable accesses are marked by instances of the `Attribute` object annotated with the type of reference: `Load`, `Store`, or `Delete`. We traverse the AST's statements in-order to build a *fine-grained* dataflow graph, where each node is a cell/statement pair, and each directed edge goes from an attribute `Load` to the corresponding `Store`(s).

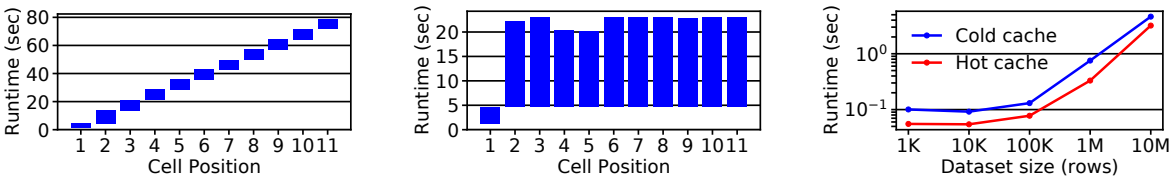
Python's scoping logic presents additional complications; first, function and class declarations may reference attributes (e.g., imports) from an enclosing scope, creating transitive dependencies. When traversing a function or class declaration, we record such dependencies and include them when the symbol is `Loaded`. Transitive dependency tracking is complicated due to Python's use of mutable closures (e.g., see Figure 3); In the latter code block, when `bar` is declared, it 'captures' the scope of `foo`, in which `a = 2`, and overrides an assignment in the global scope, even though the enclosing scope is not otherwise accessible.

Afterwards, the fine-grained dataflow graph, produced as explained above, is reduced into a *coarse-grained* dataflow graph by (i) merging nodes for the statements in a cell, (ii) removing self-edges, and (iii) removing parallel edges with identical labels. The coarse-grained data flow graph provides an approximation of the cell's dependencies: The set of in-edges (resp., out-edges) is typically an upper bound on the cells real dependencies. While missed dependencies are theoretically possible, they are rare in the type of code used in typical Jupyter notebooks. Nonetheless, if they arise they will be taken care of by our scheduler. As a final step, we inject explicit variable imports and exports (using Vizier's artifact API) for the read and write sets of each cell into the cell's code.

6 IMPLEMENTATION

As a proof of concept, we implemented the static analysis approach from Section 5 as a provenance-aware parallel scheduler (Section 4) within the Vizier notebook system [1]. Parallelizing cell execution requires an ICE architecture, which comes at the cost of increased communication overhead relative to monolithic kernel notebooks.

Implementation. The parallel scheduler was integrated into Vizier 1.2 (https://github.com/VizierDB/vizier-scala). We additionally added a pooling feature to mitigate Python's high startup cost (approximately 600ms); The modified Vizier launches a small pool of continuously running Python instances. In future work, we plan to allow kernels to cache artifacts, and prioritize the use of kernels that have already loaded artifacts we expect the cell to read. This prototype does not yet implement repairs for missed dependencies.



(a) Serial Execution **(b) Parallel Execution** **(c) Scalability - Read**

Figure 4: Workload traces for a synthetic reader/writer workload

Experiments. All experiments were run on Ubuntu 20.04 on a server with 2 x AMD Opteron 4238 CPUs (3.3Ghz), 128GB of RAM, and 4 x 1TB 7.2k RPM HDDs in hardware Raid 5. Vizier's internal dataframe representation relies on Apache Arrow and Spark. To mitigate Spark's high start-up costs, we prefix all notebooks under test with a single reader and writer cell to force initialization of e.g., Spark's HDFS module. These are not included in timing results.

Overview. As a preliminary experiment, we ran a synthetic workload consisting of one cell that randomly generates a 100k-row, 2 integer column Pandas dataframe and exports it, and 10 reader cells that read the dataset and perform a compute intensive task: Computing pairwise distance for a 10k-row subset of the source dataset. Figure 4 shows execution traces for the workload in Vizier with its default (serial) scheduler and Vizier with its new (parallel) scheduler. The experiment shows that the parallel execution is ~ 4 times faster than the serial execution. However, each individual reader takes longer to finish in the parallel execution. This is possibly the result of contention on the dataset. Nonetheless, this preliminary result and the analysis shown in Figure 1 demonstrate the potential for parallel execution of notebooks.

Scaling. Figure 4c shows the overhead of loading data into a new kernel by creating a python cell that iterates over all records in the pandas dataframe. The 'cold cache' cost also includes the one-time cost of loading a Parquet dataset and export it via Arrow.

7 RELATED WORK

Workflow provenance has been studied extensively (e.g., see [3] for a survey), but reliance on explicit dependencies limits its utility in our setting. More closely related are provenance and static analysis techniques from the programming languages community [7].

Pimentel et al. [8] provide an overview of research on provenance for scripting (programming) languages and did identify the need for and challenges of fine-grained provenance in this context. noWorkflow [9] collects several types of provenance for Python scripts including environmental information, as well as static and dynamic data- and control-flow, but in contrast to our work only produces provenance for analysis and debugging and not scheduling. [5] combines static and dynamic dataflow analysis to track dataflow dependencies during cell execution and warn users of "unsafe" interactions where a cell is reading an outdated version of a variable. By contrast, our approach automatically refreshes dependent cells. Vamsa [6] also employes static dataflow analysis to analyze provenance of Python ML pipelines. Dataflow notebooks [4] extend Jupyter with immutable identifiers for cells and the capability to reference the results of a cell by its identifier. This approach can avoid implicit dependencies, but requires users to be diligent in using these features. Additionally, our approach allows parallel execution of independent cells, something that was only alluded to as a possibility in [4]. Nodebook [12] is a plugin for Jupyter that checkpoints notebook state in between cells to force in-order cell evaluation; Although closely related to our approach, it does not attempt parallelism, nor automatic re-execution of cells. [2] captures fine-grained provenance at runtime for common classes of relational data transformations in Python preprocessing pipelines. In contrast our approach utilizes static analysis.

8 CONCLUSIONS

We introduced an approach for incrementally refining provenance for computational notebooks and implemented a scheduler for ICE-architecture notebooks based on this approach. Our method enables (i) parallel cell execution; (ii) automatic refresh of dependent cells after modifications; and (iii) import of Jupyter notebooks. While our proof-of-concept shows promise, further work is needed to reduce state transfer between kernels. For example, kernels can be re-used or forked to minimize state transfer. Alternatively, responsibility for hosting state can be moved from the coordinator, directly to the kernel that created the state. We also plan to explore how to reduce the initial dataframe access cost.

Acknowledgements. The authors would like to thank Breadcrumb Analytics for their work on Vizier, and acknowledge support from the NSF under awards IIS-1956149 and ACI-1640864.

REFERENCES

[1] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, and J. Freire. Your notebook is not crumby enough, replace it. In *CIDR*, 2020.
[2] A. Chapman, P. Missier, G. Simonelli, and R. Torlone. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *PVLDB*, 14(4):507–520, 2020.
[3] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Eng. Bull.*, 32(4):44–50, 2007.
[4] D. Koop and J. Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *TaPP*, 2017.
[5] S. Macke, A. G. Parameswaran, H. Gong, D. J. L. Lee, D. Xin, and A. Head. Fine-grained lineage for safer notebook interactions. *PVLDB*, 14(6):1093–1101, 2021.
[6] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. Vamsa: Automated provenance tracking in data science scripts. In *SIGKDD*, pages 1542–1551, 2020.
[7] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
[8] J. F. Pimentel, J. Freire, L. Murta, and V. Braganholo. A survey on collecting, managing, and analyzing provenance from scripts. *ACM Comput. Surv.*, 52(3):47:1–47:38, 2019.
[9] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. Noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB*, 10(12):1841–1844, 2017.
[10] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. Understanding and improving the quality and reproducibility of jupyter notebooks. *Empir. Softw. Eng.*, 26(4):65, 2021.
[11] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.
[12] K. Zielnicki and J. Nunez-Iglesias. Nodebook. https://github.com/stitchfix/nodebook, 2018.