DAG lakehouse planning with an ephemeral and embedded graph database

Luca Bigon
Bauplan Labs
luca.bigon@bauplanlabs.com

Jacopo Tagliabue
Bauplan Labs
jacopo.tagliabue@bauplanlabs.com

Semih Salihoğlu
Kùzu Inc.
semih@kuzudb.com


```python
@bauplan.model()
@bauplan.python("3.10", pip={"pandas": "2.0"})
def cleaned_data(
    # reference to its parent DAG node
    data=bauplan.Model(
        "raw_data",
        columns=["c1", "c2", "c3"],
        filter="eventTime BETWEEN 2023-01-01 AND 2023-02-01"
    )
):
   # the body returns a dataframe after transformations
   return data.do_something()

@bauplan.model()
@bauplan.python("3.11", pip={"pandas": "1.5"})
def final_data(
    data=bauplan.Model("cleaned_data")
):
   return data.do_something()
```

Figure 1: A two nodes DAG in Bauplan.

ABSTRACT

Bauplan is a code-first lakehouse built by vertically integrating through APIs modular data components – catalog, I/O, runtime, Flight server etc. [5]. To abstract the underlying complexity away from users, Bauplan provides a declarative functional framework [4] to express multi-language data pipelines over Iceberg tables (Fig. 1). The planner is a pluggable module taking as input user code, and producing a *logical plan* with the DAG topology (Fig. 2, top). The planner then maps declarative user instructions to platform operations, finalizing the *physical plan* (Fig. 2, bottom) needed by workers for containerized execution [3].

Characteristically, the planner needs to perform static inferences over functional DAGs (with opaque nodes), resembling both databases and FaaS schedulers. Similar to database planners, Bauplan's planner combines filters for efficient I/O scans, and validates column matching of adjacent nodes. Similar to FaaS planners, it unifies Python packages along the transitive dependency graph, and infers function ordering from their signature. *We present a graph-based planning module that uses an embedded graph database management system (GDBMS) – Kuzu [1] – in a novel way*:

1. User code is parsed and inserted into an ephemeral database in Kuzu, which represents both data (e.g. the required Iceberg push downs) and runtime entities (e.g. required Python packages).
2. We execute static checks and planning steps on the graph using Cypher queries (e.g. do all children functions have a parent? Add one *S3 read* node before function f).
3. The final graph is serialized into Protobuf for execution by downstream workers, and then is destroyed.

Our initial planner was a home-grown Python library with imperative recursive functions for inference and static checks, which was both slow and error-prone. Instead, following the philosophy of composable data systems [2], we chose to utilize a GDBMS that gave us: (i) a high-level query language (Cypher), simplifying our DAG inference through recursive queries; (ii) optimized query execution with leveraging multi-core hardware.

Given the on-demand nature of our workloads, we wish to move the embedded GDBMS in memory, further simplifying our infrastructure life-cycle and speeding up queries. In collaboration with the Kuzu team, we developed an in-memory version of their database, so that we could leverage a new, ephemeral graph at every run: as a result, we currently create tens of thousands of ephemeral graph databases on-the-fly per day (and growing). The in-memory version provided optimized inference without infrastructure dependencies, updates to our build system, or changes in the life-cycle of user requests. Today, a single request may involve >500 Cypher statements (between entity creation, pattern matching, graph updates), which are all executed with sub-second latency.

We obtained a (on average) 20x faster planner compared to our original solution, with composability also improving engineering efficiency and debuggability [2]: since the DAG plan is now expressed in a Python-agnostic representation, it can be dumped, inspected, tested and visualized at any moment, without depending on the rest of the distributed system. While our planning needs are lakehouse-oriented, we believe our solution to be of broader interest since graphs are a natural representations for many states in data systems.

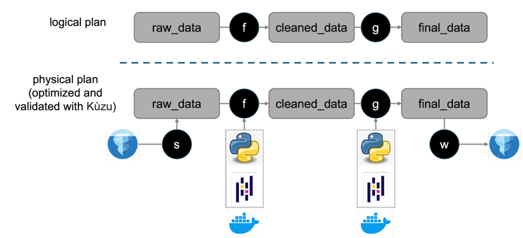

Figure 2: The logical plan is created by parsing user code, the physical plan is obtained running Cypher on Kuzu.

REFERENCES

[1] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu Graph Database Management System. In *The Conference on Innovative Data Systems Research*.

[2] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *PVLDB* 16, 10 (June 2023), 2679–2685.

[3] Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. 2024. Bauplan: Zero-copy, Scale-up FaaS for Data Pipelines. In *WoSC*.

[4] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing. In *2024 IEEE International Conference on Big Data (BigData)*. 3562–3567. https://doi.org/10.1109/BigData62323.2024.10825377

[5] Jacopo Tagliabue, Ciro Greco, and Luca Bigon. 2023. Building a Serverless Data Lakehouse from Spare Parts. *ArXiv* abs/2308.05368 (2023).

Proceedings of the VLDB Endowment. ISSN 2150-8097.