Lecture #21: System Analysis (Yellowbrick)

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction to Yellowbrick

Yellowbrick [1] is an OLAP DBMS written in C++. It originated from a hard fork of PostgreSQL v9.5 and initially functioned as an on-prem appliance with FPGA acceleration, offering a hardware-optimized solution. Transitioning to DBaaS in 2021, it now utilizes Kubernetes to orchestrate its microservices-based architecture. Designed for high performance and elasticity in both public and private clouds, Yellowbrick supports dynamic scaling and simplifies configuration with a SQL interface over Kubernetes. The architecture facilitates rapid deployment and emphasizes performance enhancements in networking and storage to boost query efficiency and responsiveness.

2 Design

This section provides a discussion of Yellowbrick's design choices, covering its architectural framework, query execution, compilation, optimization, and storage layers.

2.1 High-Level Overview

This subsection highlights Yellowbrick's main design decisions:

1. It uses a shared-disk / disaggregated storage architecture.
2. It adopts a push-based vectorized query processing model.
3. It supports transpilation query code generation in C++.
4. It makes heavy use of compute-side caching to reduce accesses to the disaggregated storage.
5. It has a separate row and custom PAX columnar storage format.
6. The query optimizer is built on PostgreSQL's query optimizer with custom heurstics and improvements.
7. It supports both Sort-Merge Join and Hash Join operations.
8. It has a large number of low-level systems optimizations.

2.2 Architectural Design

From a high level, Yellowbrick comprises three main components:

- **A data warehouse instance** that serves as the front-end, managing connections, query parsing, plan caching, row storage, metadata, and concurrency control.
- **Worker nodes** responsible for query execution, compute hardware management, and local cache maintenance.
- **Background/Maintenance Nodes** tasked with compilation and bulk loading.

These components are structured as microservices within Docker pods, managed by Kubernetes, which ensures system state management, scalability, and provisioning. Kubernetes also conceals its operations behind SQL, and the system designates one worker pod per node to ensure exclusive hardware access.

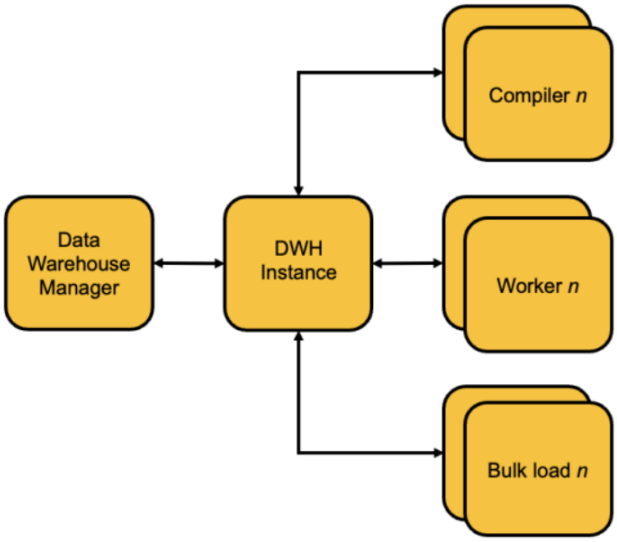

Figure 1: Microservices in Yellowbrick

2.3 Query Execution

Yellowbrick utilizes push-based vectorized query processing that accommodates both row- and columnar-oriented data formats, incorporating early materialization to structure data immediately after retrieval. This system allows for immediate access to complete data tuples or rows. Additionally, it features transpose operators for converting data between row and PAX columnar storage formats, aiming to ensure that data processing occurs within the CPU's L3 cache to minimize memory accesses.

2.4 Query Compilation/Transpilation

Yellowbricks enhances query performance through holistic query compilation, which involves transpiling query plans into C++ code via source-to-source translation. Initially, it divides query plans into independent fragments, each of which is then converted into C++ source code. These fragments are compiled into machine code using LLVM by a dedicated compilation service. At runtime, dynamic linking is used to stitch these fragments back together. This fragmentation strategy increases the reusability of code, as smaller fragments are more likely to be utilized in subsequent queries.

2.5 Query Optimizer

Yellowbrick leverages a heavily modified version of PostgreSQL's stratified optimizer to enhance its query execution capabilities. It incorporates advanced zone map filtering and employs a cost-based join order selection that utilizes detailed statistics. These statistics are derived from processes such as rowstore compaction and thorough ANALYZE passes, which help in gathering valuable data insights including histograms and HyperLogLog summaries. Additionally, Yellowbrick enhances the efficiency of hash joins through the innovative use of sideways information passing of Bloom filters, optimizing data retrieval and processing across its architecture.

2.6 Storage

Yellowbrick utilizes a proprietary file format for its managed storage system, allowing specific attributes such as sharding or local-sorting to be set for each table. The storage format organizes data into approximately 100MB files, subdivided into 2MB chunks, optimizing the system for efficient data access and management. It supports bulk loading of Parquet files, albeit with some limitations. Yellowbrick also main-

tains a dual storage approach: keeping row-store data on the front-end and transferring it to columnar files in the object store through background tasks. This setup enhances performance by enabling compaction of modified columnar files and allowing direct bulk loads to the object store, bypassing the row-store and worker SSD caches for increased efficiency.
In Yellowbrick's database management system, data file assignment to workers utilizes Rendezvous Hashing, a method also employed by systems like Druid and Ignite. This hashing technique involves generating a hash value for each worker by appending the worker's identifier to a hashed key of the file. Among these hash values, the one with the highest weight is selected to determine the worker that will handle the file. This approach ensures a balanced and efficient distribution of data files across the available workers.

3 OS Optimizations

The following section discusses the OS-level optimizations implemented in Yellowbrick, including memory allocation, cooperative thread scheduler, and device drivers.

3.1 Memory Allocation

Yellowbrick incorporates a custom, NUMA-aware, latch-free memory allocator that secures all required memory at startup using mmap with mlock to utilize huge pages, enhancing efficiency and performance. This method groups allocations by query to minimize fragmentation and is claimed to be 100 times faster than standard libc malloc. Additionally, each worker is equipped with a buffer pool manager that adopts a MySQL-style approximate LRU-K algorithm for managing cached data files, optimizing memory usage and data retrieval processes. (Sidenote: The LRU-K algorithm is a caching mechanism that extends the traditional Least Recently Used (LRU) approach by considering the last K references to each item in a cache. Unlike basic LRU, which evicts the least recently accessed item, LRU-K tracks and evaluates the historical access patterns up to K instances for each item, allowing it to make more informed decisions on which items to evict.)

3.2 Scheduler

Yellowbrick employs a custom cooperative multi-tasking thread scheduler, using coroutines that synchronize every 100 milliseconds with a centralized cluster scheduler. This scheduling approach ensures that only one query executes at a time across the entire cluster, with all cores on a worker performing the same task simultaneously. This strategy aims to optimize data processing by ensuring that cores handle data that is currently in the CPU's L3 cache, rather than retrieving it from memory, thereby enhancing performance and efficiency.

3.3 Device Drivers

Yellowbrick utilizes custom NVMe and NIC drivers that operate in user-space to minimize memory copy overheads, with the capability to fallback to Linux drivers when necessary. Additionally, it employs a custom reliable UDP network protocol that bypasses the kernel using DPDK for internal communications, enhancing efficiency. This setup includes dedicated receive and transmit queues for each CPU, which are polled asynchronously to manage internal data flow. The system ensures that data is only sent to a designated "partner" CPU on other workers, optimizing network communication and resource utilization.

4 Conclusion

Yellowbrick's system engineering capabilities are undeniably impressive, showcasing advanced use of technologies such as custom NVMe/NIC drivers and a reliable UDP network protocol through DPDK to opti-

Cluster Size	Runtime with TCP	Runtime with DPDK	Speedup due to DPDK
2	2430s	1976s	19%
3	1626s	1358s	17%
4	1222s	995s	19%

Figure 2: Sequential runtime of the 99 TPC-DS queries at 1 TB scale versus compute cluster size and network implementation

mize performance. If one were to build similar systems today, eBPF might be a more modern choice over DPDK. However, it's crucial to note that the efficacy of all these technological optimizations hinges on the database management system's ability to generate efficient query plans. Without smart query planning, even the most sophisticated system optimizations may fail to deliver expected performance improvements.

References

[1] M. Cusack, J. Adamson, M. Brinicombe, N. Carson, T. Kejser, J. Peterson, A. Vasudev, K. Westerfeld, and R. Wipfel. Yellowbrick: An elastic data warehouse on kubernetes.