ADVANCED DATABASE SYSTEMS

Yellowbrick Database System

21

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

ADMINISTRIVIA

Project:

→ Final Presentations: **Thursday May 2nd @ 9:00am**
→ See Piazza@59 for more information.

Final Exam:

→ Given in class on **Wednesday April 24th**
→ Due on the same day as Final Presentation

LAST CLASS

DuckDB Embedded OLAP DBMS

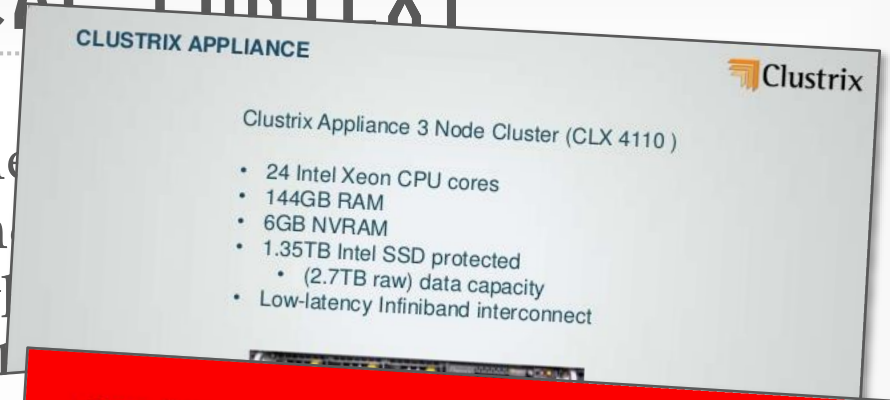
HISTORICAL CONTEXT

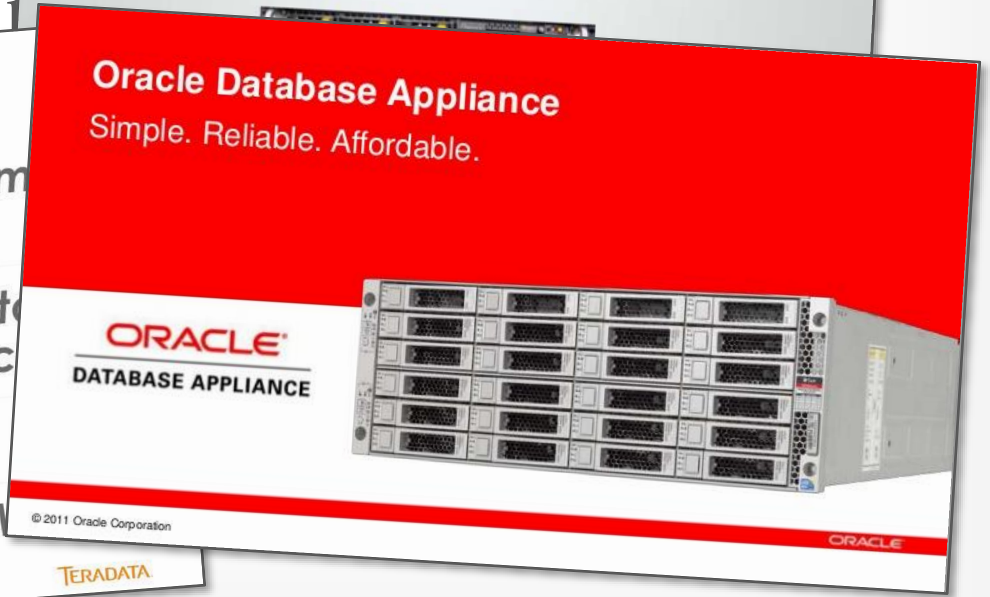
Most DBMSs are designed for off-the-shelf hardware. But some vendors sell a complete solution ("appliance") where the DBMS is optimized for a specific hardware configuration.

→ Companies also fab custom database accelerators too.

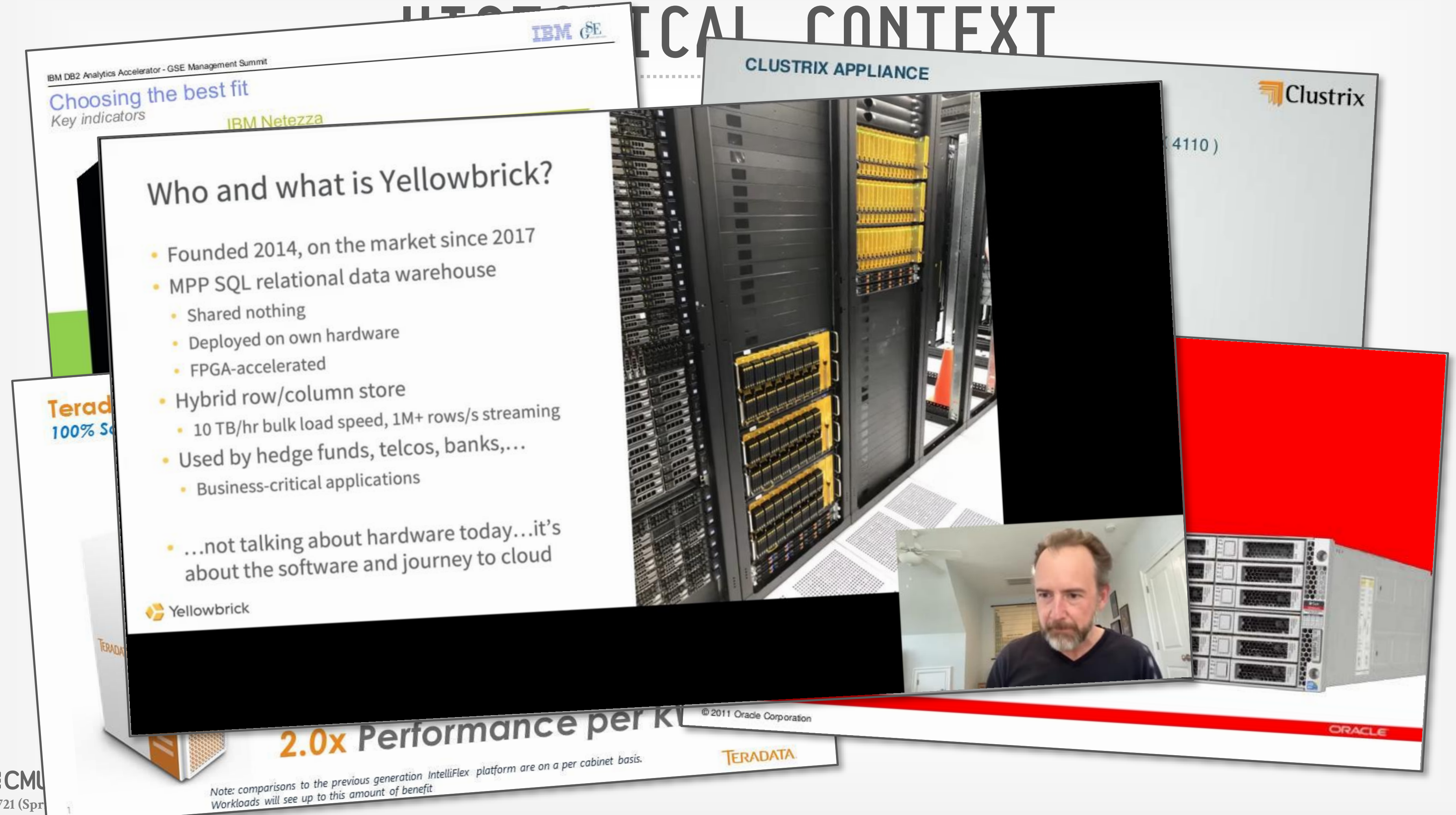
Yellowbrick started off as a high-end OLAP appliance…

HISTORICAL CONTEXT

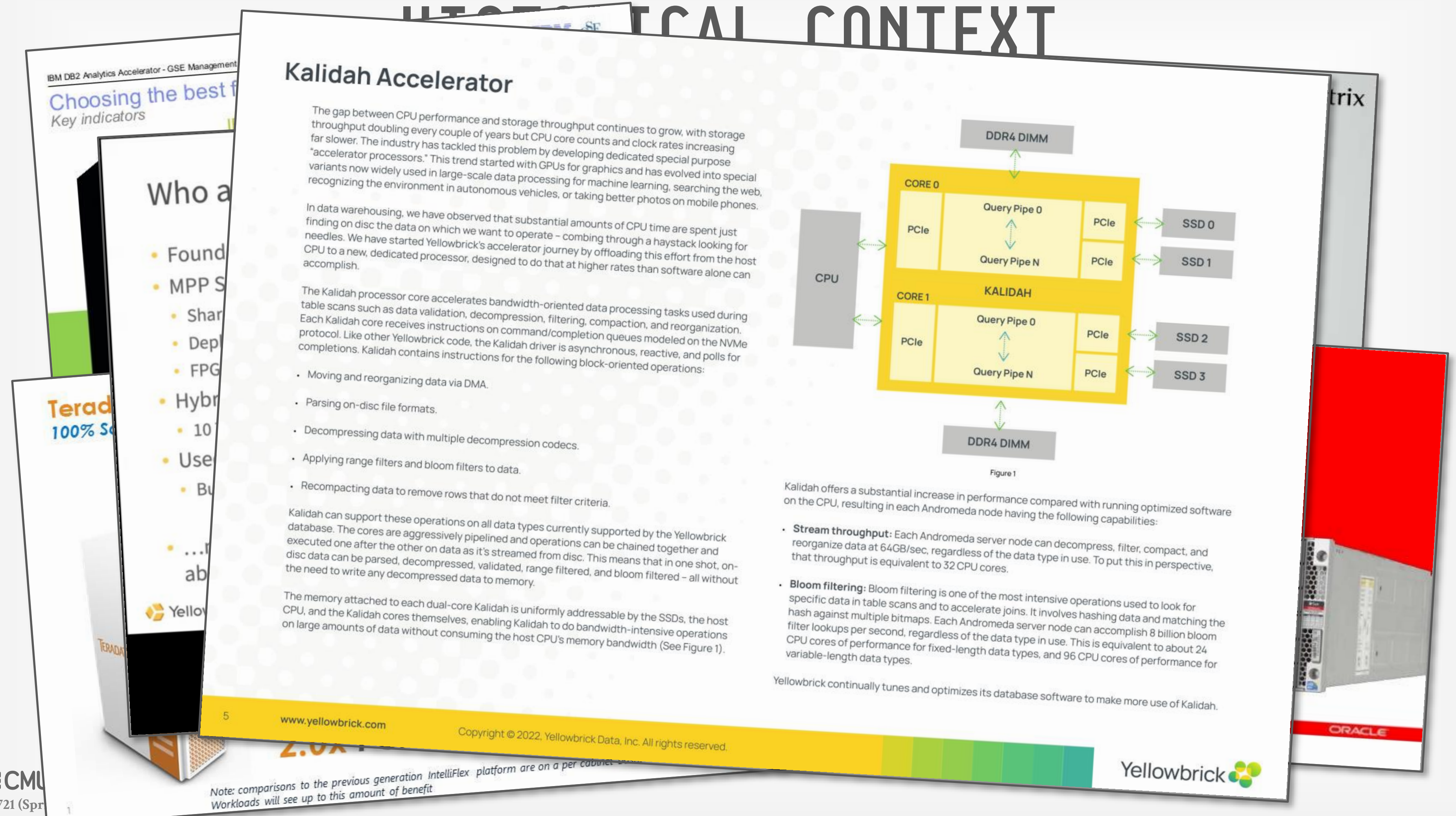




ICAL CONTEXT



HISTORICAL CONTEXT



Kalidah Accelerator

The gap between CPU performance and storage throughput continues to grow, with storage throughput doubling every couple of years but CPU core counts and clock rates increasing far slower. The industry has tackled this problem by developing dedicated special purpose "accelerator processors." This trend started with GPUs for graphics and has evolved into special variants now widely used in large-scale data processing for machine learning, searching the web, recognizing the environment in autonomous vehicles, or taking better photos on mobile phones.

In data warehousing, we have observed that substantial amounts of CPU time are spent just finding on disc the data on which we want to operate – combing through a haystack looking for needles. We have started Yellowbrick's accelerator journey by offloading this effort from the host CPU to a new, dedicated processor, designed to do that at higher rates than software alone can accomplish.

The Kalidah processor core accelerates bandwidth-oriented data processing tasks used during table scans such as data validation, decompression, filtering, compaction, and reorganization. Each Kalidah core receives instructions on command/completion queues modeled on the NVMe protocol. Like other Yellowbrick code, the Kalidah driver is asynchronous, reactive, and polls for completions. Kalidah contains instructions for the following block-oriented operations:

- Moving and reorganizing data via DMA.
- Parsing on-disc file formats.
- Decompressing data with multiple decompression codecs.
- Applying range filters and bloom filters to data.
- Recompacting data to remove rows that do not meet filter criteria.

Kalidah can support these operations on all data types currently supported by the Yellowbrick database. The cores are aggressively pipelined and operations can be chained together and executed one after the other on data as it's streamed from disc. This means that in one shot, on-disc data can be parsed, decompressed, validated, range filtered, and bloom filtered – all without the need to write any decompressed data to memory.

The memory attached to each dual-core Kalidah is uniformly addressable by the SSDs, the host CPU, and the Kalidah cores themselves, enabling Kalidah to do bandwidth-intensive operations on large amounts of data without consuming the host CPU's memory bandwidth (See Figure 1).

Figure 1

Kalidah offers a substantial increase in performance compared with running optimized software on the CPU, resulting in each Andromeda node having the following capabilities:

- **Stream throughput:** Each Andromeda server node can decompress, filter, compact, and reorganize data at 64GB/sec, regardless of the data type in use. To put this in perspective, that throughput is equivalent to 32 CPU cores.
- **Bloom filtering:** Bloom filtering is one of the most intensive operations used to look for specific data in table scans and to accelerate joins. It involves hashing data and matching the hash against multiple bitmaps. Each Andromeda server node can accomplish 8 billion bloom filter lookups per second, regardless of the data type in use. This is equivalent to about 24 CPU cores of performance for fixed-length data types, and 96 CPU cores of performance for variable-length data types.

Yellowbrick continually tunes and optimizes its database software to make more use of Kalidah.

5 www.yellowbrick.com

Yellowbrick

YELLOWBRICK (2014)

OLAP DBMS written on C++ and derived from a hardfork of PostgreSQL v9.5.

→ Uses PostgreSQL's front-end (networking, parser, catalog) to handle incoming SQL requests.

Originally started as an on-prem appliance with FPGA acceleration. Switched to DBaaS in 2021.

Cloud-version uses Kubernetes for all components.

YELLOWBRICK: AN ELASTIC DATA WAREHOUSE ON KUBERNETES
CIDR 2024

YELLOWBRICK

Shared-Disk / Disaggregated Storage

Push-based Vectorized Query Processing

Transpilation Query Codegen (C++)

Compute-side Caching

Separate Row + PAX Columnar Storage

Sort-Merge + Hash Joins

PostgreSQL Query Optimizer++

Insane Systems Engineering

YELLOWBRICK: ARCHITECTURE

Data Warehouse Instance:

→ Front-end service that manages connections, parsing, plan caching, row store, meta-data, and concurrency control.

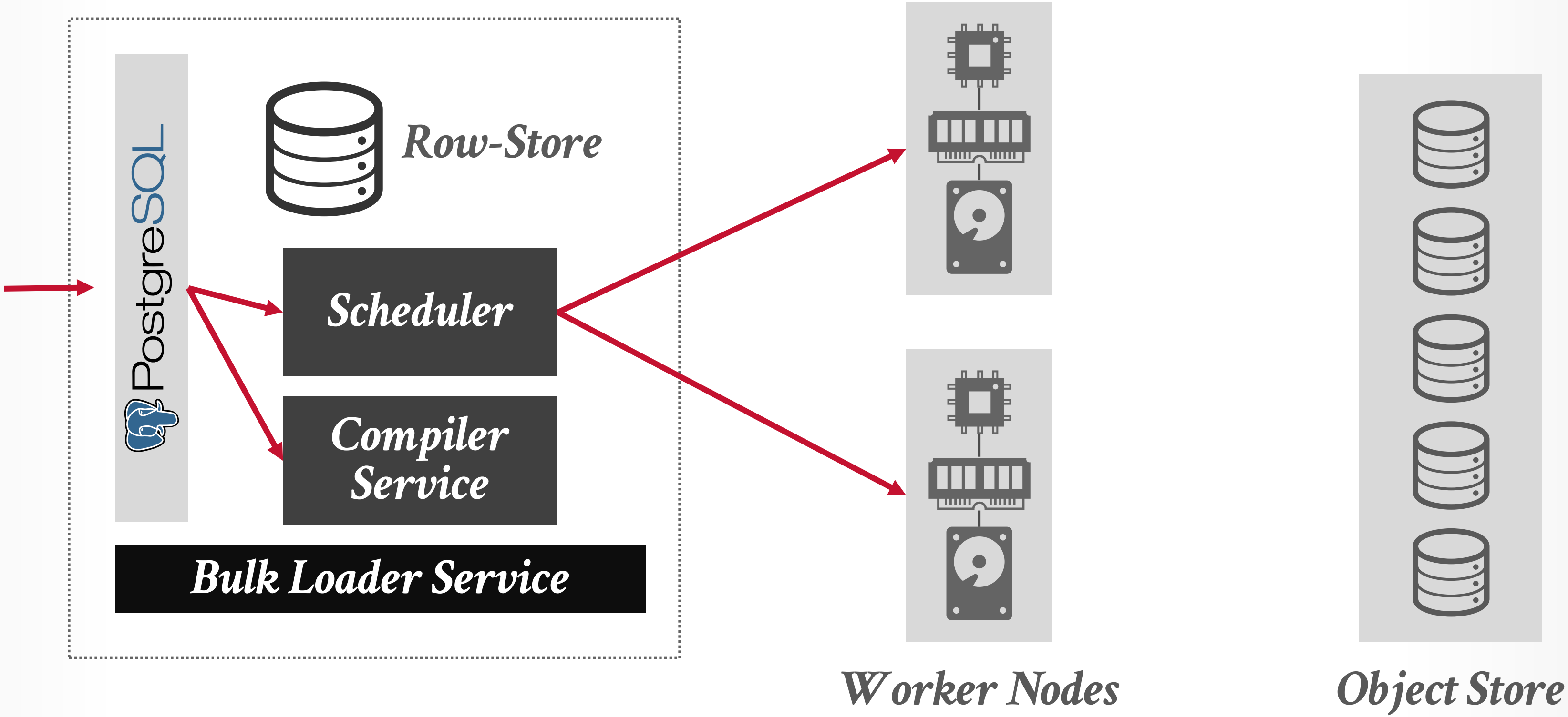
Worker Nodes:

→ Responsible for query execution, managing compute hardware, and maintaining local cache.

Background / Maintenance Nodes:

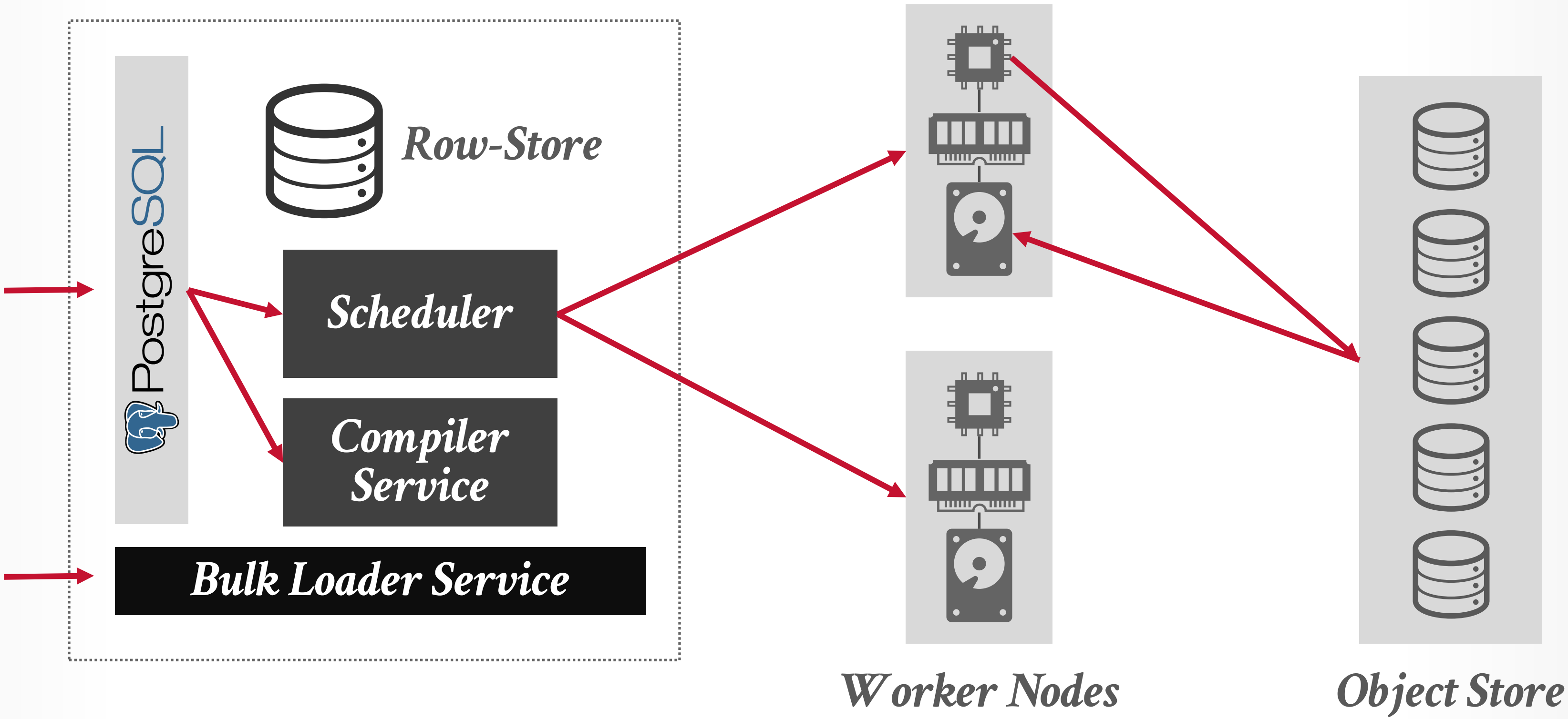
→ Compilation, Bulk Loading

YELLOWBRICK: ARCHITECTURE



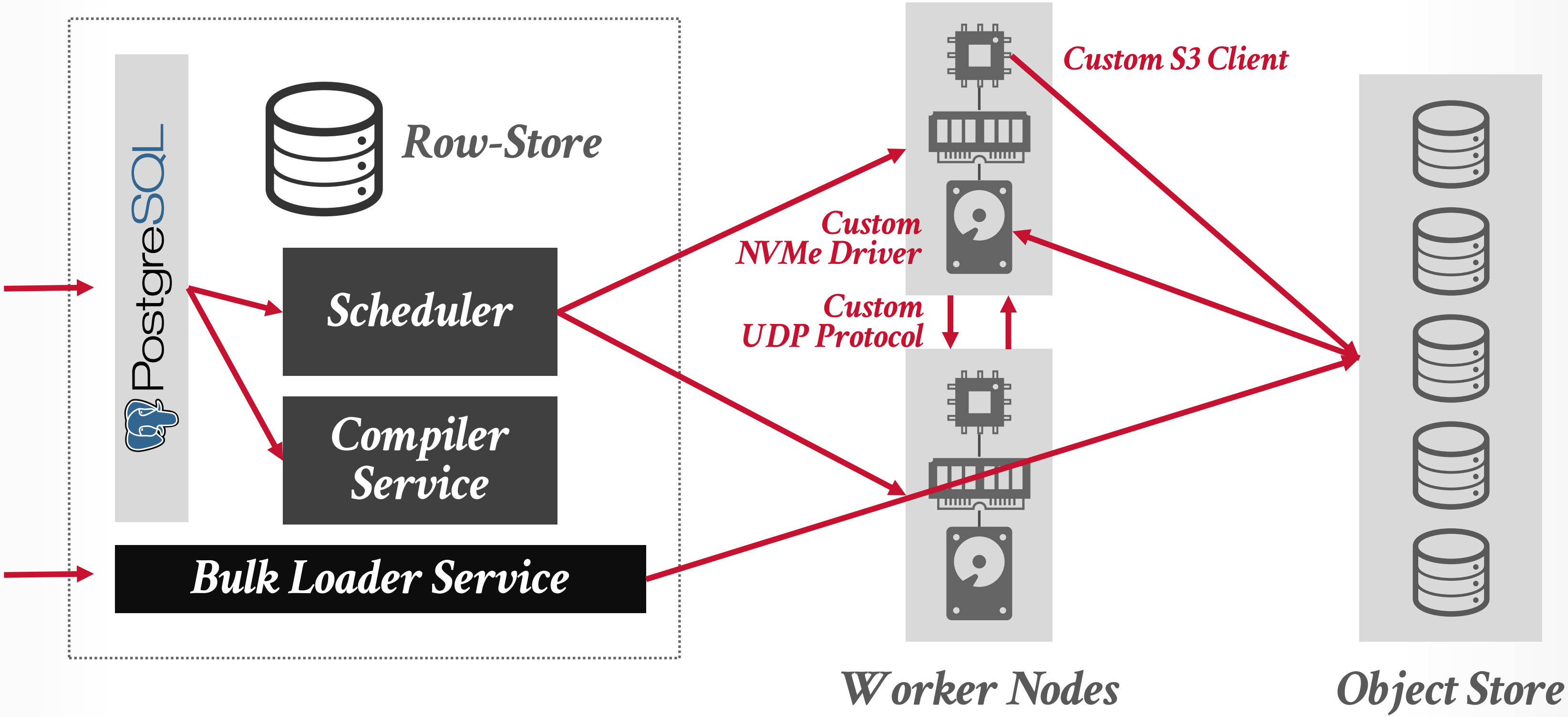
Source: Mark Cusack

YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

YELLOWBRICK: ARCHITECTURE

Based on a microsevice architecture where all components run as Docker pods in Kubernetes.

→ Kubernetes handles system state management, scalability, and provisioning.

→ Hides all Kubernetes operations behind SQL (!!!).

Assigns one worker pod per worker node to guarantee exclusive access to hardware.

YELLOWBRICK: QUERY EXECUTION

Pushed-based vectorized query processing that supports both row- and columnar-oriented data with early materialization.

→ Introduces transpose operators to convert data back and forth between row and columnar formats.

Holistic query compilation via source-to-source transpilation.

Yellowbrick's architecture goal is for workers to always process data residing in the CPU's L3 cache and not memory.

YELLOWBRICK: QUERY COMPILATION

Split query plan into independent fragments and then transpile each fragment into C++ source code.

Dedicated compilation service uses LLVM to compile each fragment into machine code.

→ Use separate threads to compile fragments and then stitch them back together at runtime with dynamic linking.

Compiler service maintains a fragment cache to reduce compilation costs.

→ Tracks engine version and other dependencies.

YELLOWBRICK: QUERY OPTIMIZER

Heavily modified version of PostgreSQL's stratified optimizer to do support zone map filtering.

Yellowbrick's main addition is a cost-based join order selection using statistics collected from row-store compaction and **ANALYZE** passes over data.

→ Histograms, HyperLogLog, HeavyHitters

Supports sideways information passing of Bloom filters for hash joins.

Source: Mark Cusack

YELLOWBRICK: STORAGE

Yellowbrick only supports managed storage based on its proprietary file format.

- → Can specify sharding / local-sorting attribute per table.
- → ~100MB files with 2MB chunks.
- → Supports bulk loading Parquet files with some limitations.

Maintains row-store data in front-end and columnar data in object store.

- → Background task to move row-store data to columnar files.
- → Also supports compaction of modified columnar files.
- → DBMS bulk loads to object store in columnar files, bypassing row-store and worker SSD caches.

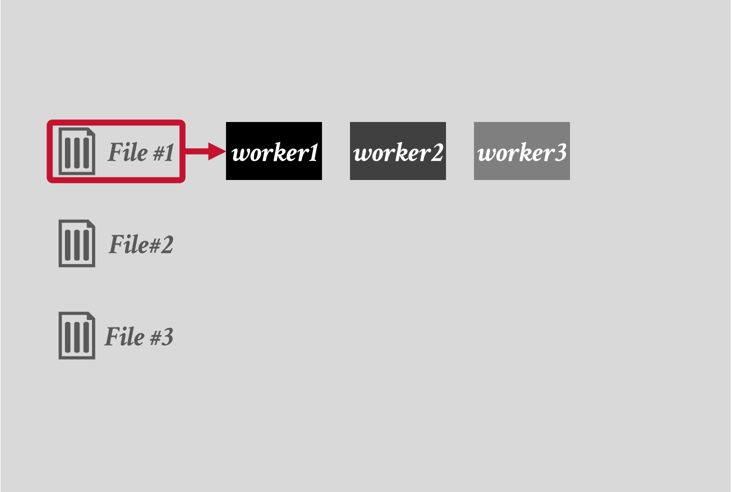
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



hash(file1 + worker1) = 100

hash(file1 + worker2) = 90

hash(file1 + worker3) = 80

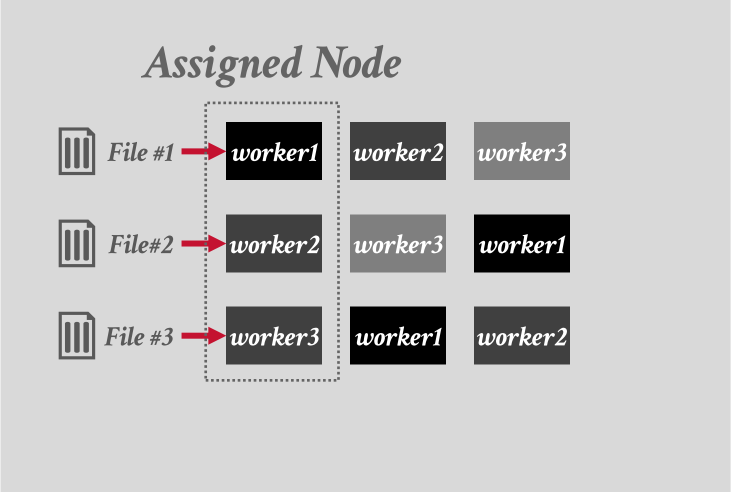
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



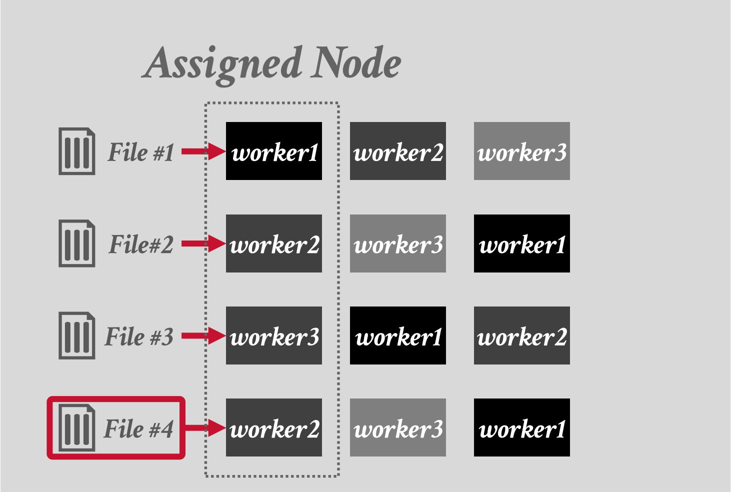
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



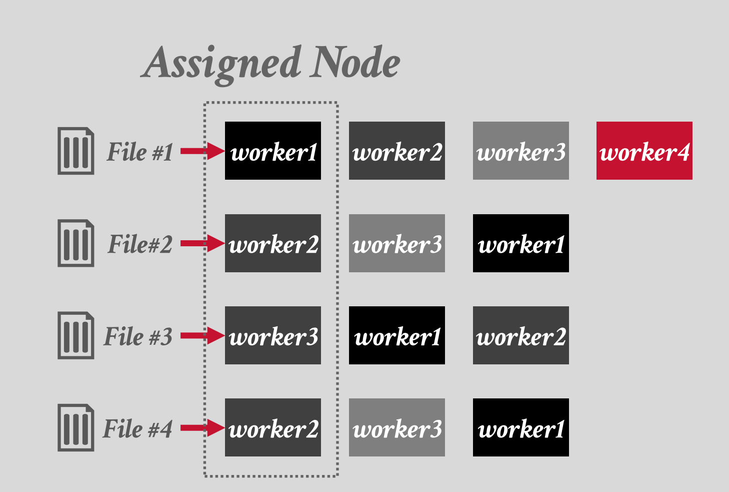
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



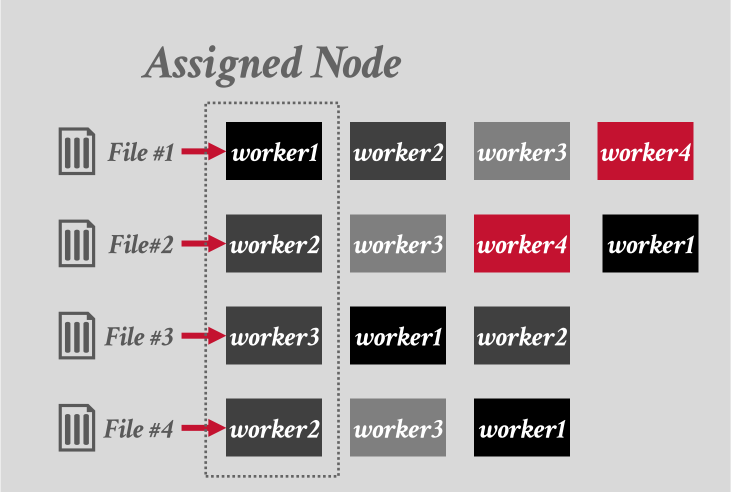
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



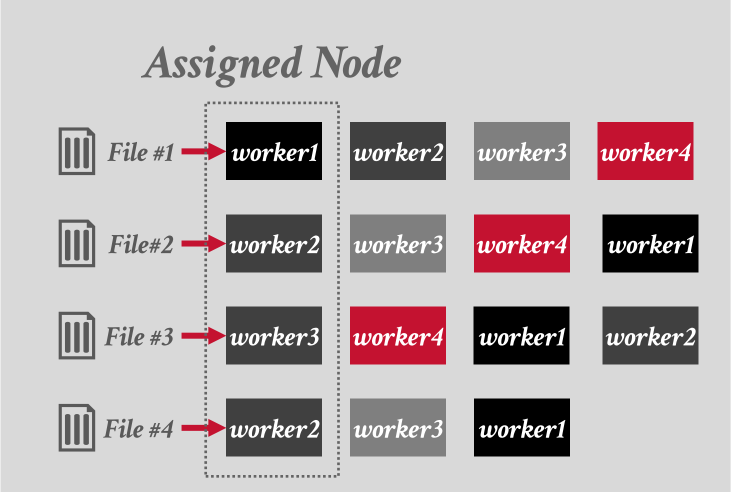
YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.

→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.

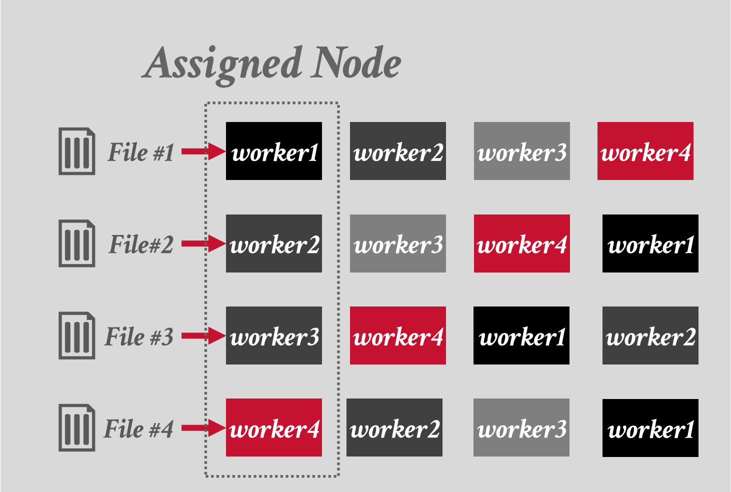


YELLOWBRICK: SHARDING

The DBMS assigns data files to workers using Rendezvous Hashing.
→ Also used in Druid and Ignite.

For each file, generate a hash value for each worker by concatenating the worker's identifier to the hashed key.

Pick the hash value with the highest weight.



OBSERVATION

Remember that the OS is our enemy.

What can a DBMS implement for itself if it wants to ensure that it never has to talk to the OS after starting up?

YELLOWBRICK: OS OPTIMIZATIONS

Memory Allocator

Thread Scheduler

Device Drivers

Network Protocols

YELLOWBRICK: MEMORY ALLOCATOR

Custom NUMA-aware, latch-free allocator that gets all the memory needed upfront at start-up

→ Using **mmap** with **mlock** with huge pages.
→ Allocations are grouped by query to avoid fragmentation.
→ Claims their allocator is 100x faster than libc **malloc**.

Each worker also has a buffer pool manager that uses MySQL-style approximate LRU-K to store cached data files.

MEMORY PAGES

OS maps physical pages to virtual memory pages.

The CPU's MMU maintains a TLB that contains the physical address of a virtual memory page.

→ The TLB resides in the CPU caches.
→ It cannot obviously store every possible entry for a large memory machine.

When you allocate a block of memory, the allocator keeps that it aligned to page boundaries.

HUGE PAGES

Instead of always allocating memory in 4 KB pages, Linux supports creating larger pages (2MB to 1GB)
→ Each page must be a contiguous blocks of memory.
→ Greatly reduces the # of TLB entries

Recent research from Google suggests that huge pages improved their data center workload by 7%.
→ 6.5% improvement in Spanner's throughput

Huge Pages makes sense in an OLAP DBMS that is accessing large read-only data blocks at a time.

Source: Evan Jones

HUGE PAGES

Instead of always allocating memory in 4 KB pages, Linux supports creating larger pages (2MB to 1GB)

→ Each page must be a contiguous blocks of memory.

→ Greatly reduces the # of TLB entries

Recent research from Google suggests that huge pages improved their data center workload by 7%.

→ 6.5% improvement in Spanner's throughput

Huge Pages makes sense in an OLAP DBMS that is accessing large read-only data blocks at a time.

Source: Evan Jones

TRANSPARENT HUGE PAGES

WARNING: THIS IS DATABASE CANCER

With **Transparent Huge Pages** (THP), the OS reorganizes and compacts pages in the background.

→ Split larger pages into smaller pages.
→ Combine smaller pages into larger pages.
→ Can cause the DBMS process to stall on memory access.

Nearly every DBMS advises to disable THP:

→ Oracle, SingleStore, NuoDB, MongoDB, Sybase, TiDB.
→ Vertica says to enable THP only for newer Linux distros.

Source: Alexandr Nikitin

TRANSPARENT HUGE PAGES

WARNING: THIS IS DATABASE CANCER

With **Trans**[illegible]**nt H**[illegible]**ges** ([illegible]), the OS reorganizes and c[illegible] the background.

- → Split larger page[illegible]
- → Combine small[illegible]ages i[illegible]ger pa[illegible].
- → Can cause the [illegible]MS proc[illegible] to stall o[illegible]memory access.

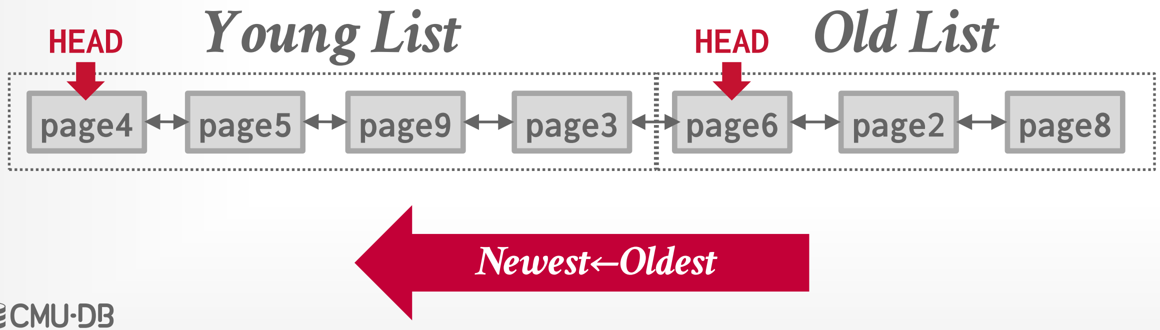
Nearly every DB[illegible]ses[illegible]ble THP:

- → Oracle, SingleStore, [illegible]DB, Sybase, TiDB.
- → Vertica says [illegible]ble [illegible]or n[illegible]inux distros.

Source: Alexandr Nikitin

MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").
- → New pages are always inserted to the head of the old list.
- → If pages in the old list is accessed again, then insert into the head of the young list.

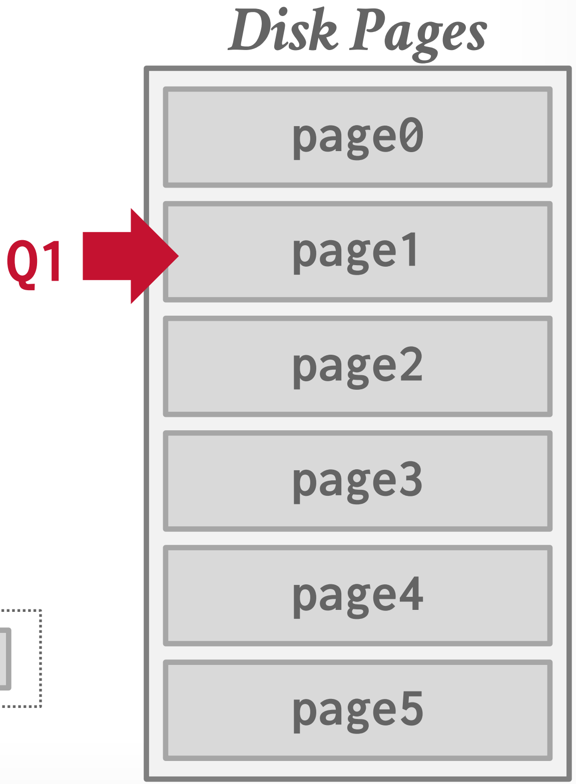


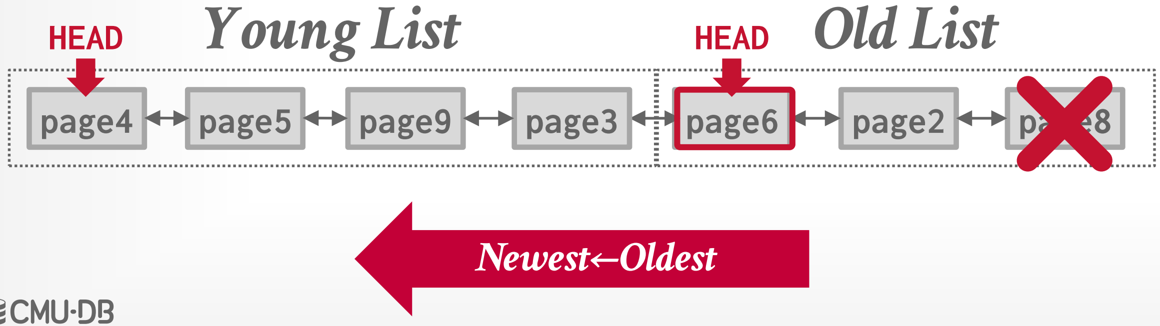
MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

→ New pages are always inserted to the head of the old list.

→ If pages in the old list is accessed again, then insert into the head of the young list.



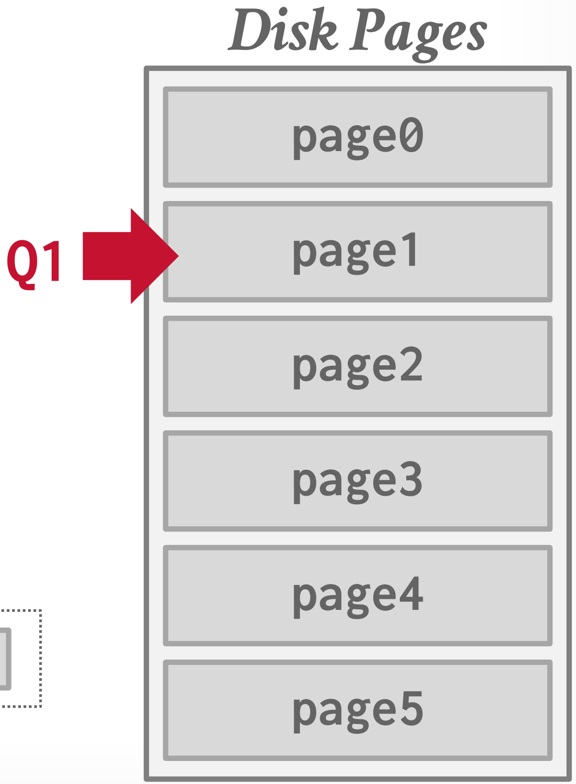


MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

→ New pages are always inserted to the head of the old list.

→ If pages in the old list is accessed again, then insert into the head of the young list.

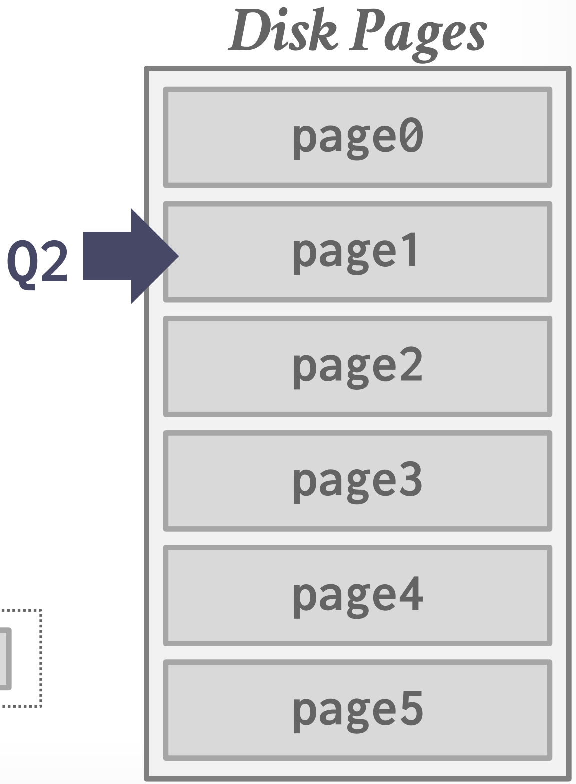


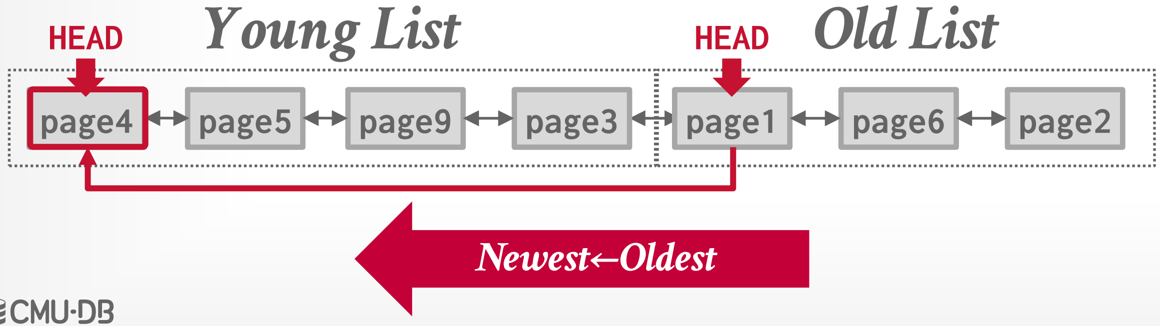
MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

→ New pages are always inserted to the head of the old list.

→ If pages in the old list is accessed again, then insert into the head of the young list.



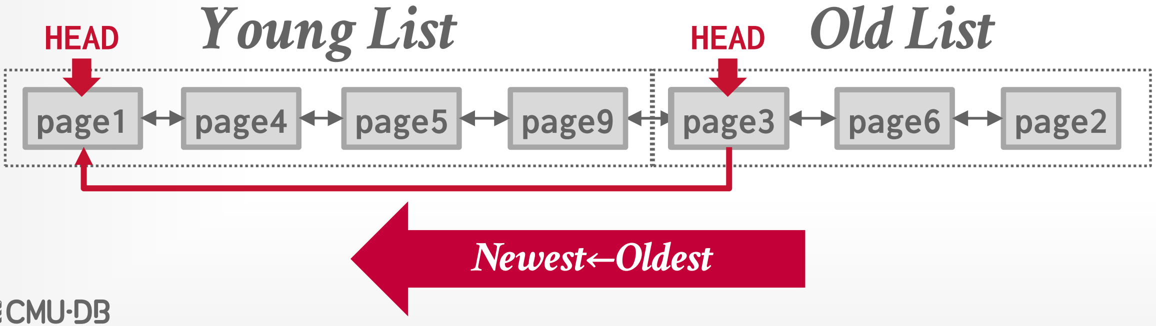


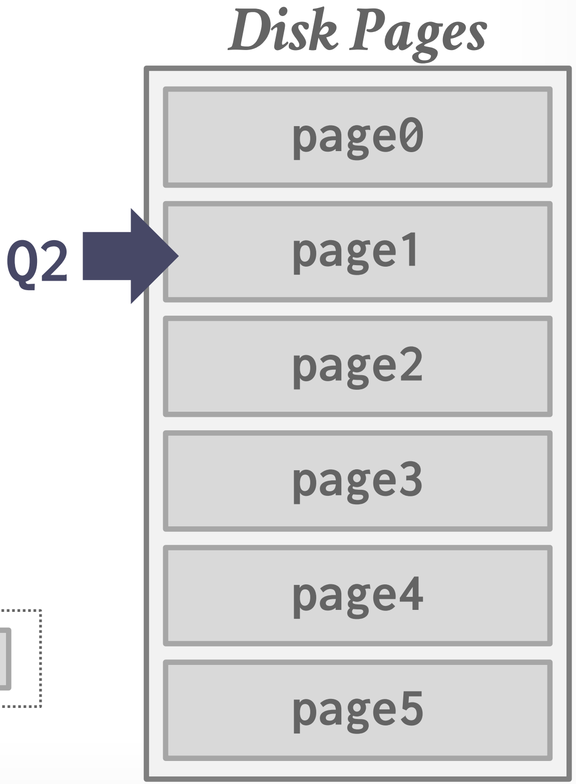
MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

→ New pages are always inserted to the head of the old list.

→ If pages in the old list is accessed again, then insert into the head of the young list.





YELLOWBRICK: SCHEDULER

Custom cooperative multi-tasking thread scheduler (coroutines) that synchronizes every 100ms with a centralized cluster scheduler.

Only one query executes at a time in a cluster. All cores on the same worker execute the same task at the same time.

→ The goal is to ensure that cores are processing recently arrived data in L3 instead of memory.

YELLOWBRICK: DEVICE DRIVERS

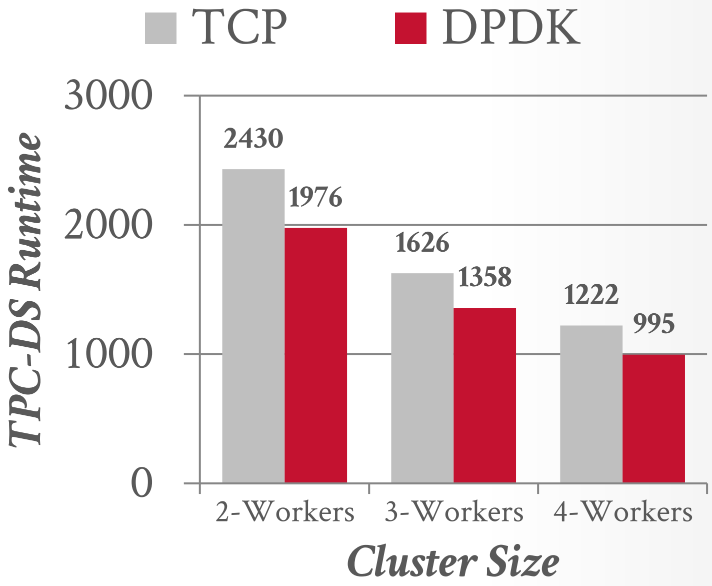
Custom NVMe / NIC drivers that run in user-space to avoid memory copy overheads.

→ Falls back to Linux drivers if necessary.

Custom reliable UDP network protocol with kernel-bypass (DPDK) for internal communication.

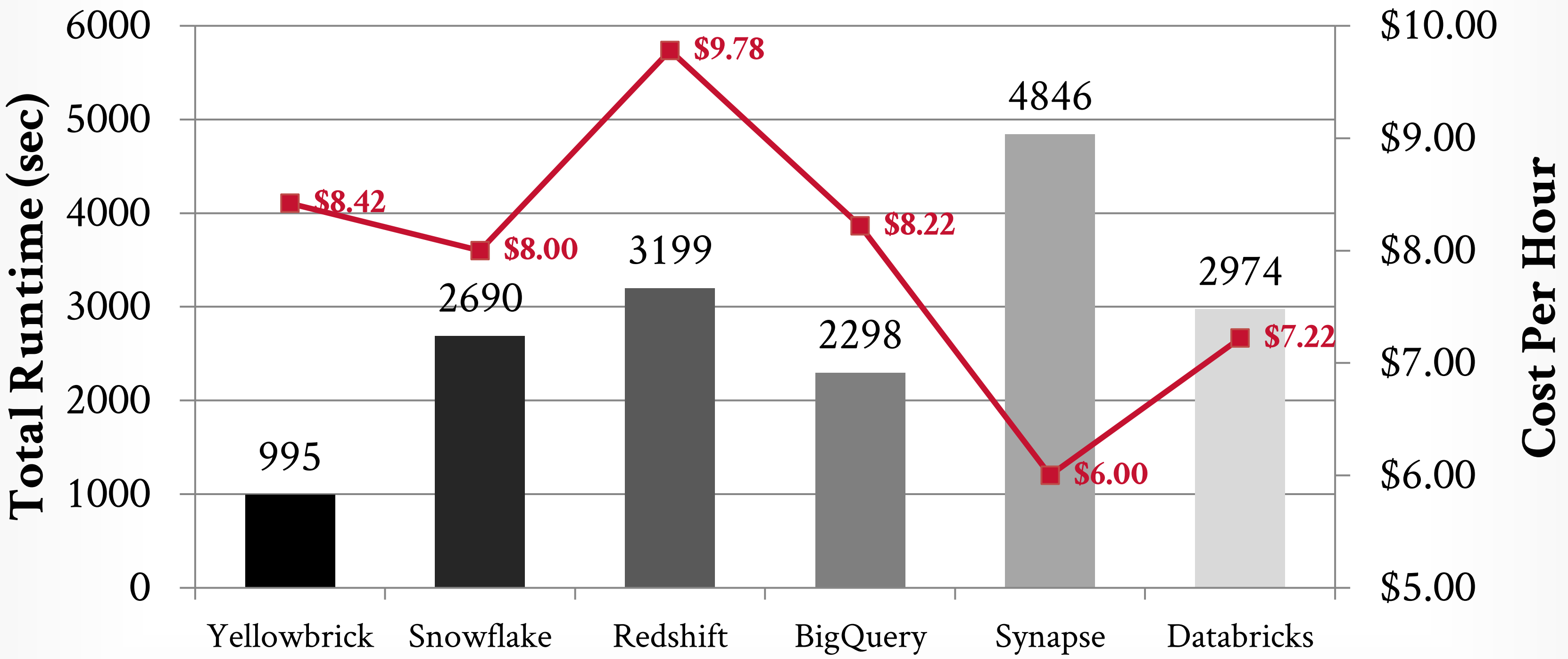
→ Each CPU has its own receive/transmit queues that it polls asynchronously.

→ Only sends data to a "partner" CPU at other workers.



BENCHMARK

TPC-DS (Scalefactor 1)



Yellowbrick	Snowflake	Redshift	BigQuery	Synapse	Databricks
$2.33	**$5.98**	**$8.69**	**$5.25**	**$8.08**	**$5.96**

Source: Mark Cusack

PARTING THOUGHTS

Yellowbrick's systems engineering street skills are ridiculously impressive.

→ If building it today, you should probably use eBPF instead of DPDK.

But remember that all these optimizations will not matter if the DBMS chooses crappy query plans.

PARTING THOUGHTS

Yellowbrick's systems engineering street skills are ridiculously impressive.

→ If building it today, you should probably use eBPF instead of DPDK.

But remember that all these optimizations will not matter if the DBMS chooses crappy query plans.

NEXT CLASS

Last lecture: Amazon Redshift