Yellowbrick: An Elastic Data Warehouse on Kubernetes

Mark Cusack, John Adamson, Mark Brinicombe, Neil Carson,
Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, Robert Wipfel
Yellowbrick Data
Mountain View, California, USA
firstname.lastname@yellowbrick.com

ABSTRACT

The Yellowbrick Data Warehouse delivers efficient, scalable and resilient data warehousing in public clouds and in private data centers. The database management system is composed of a set of Kubernetes-orchestrated microservices. Kubernetes provides the single-source-of-truth for system configuration and state, and manages all data warehouse lifecycle operations, including the creation, expansion, contraction and destruction of elastic compute resources and shared services. The common runtime provided by Kubernetes enabled us to port to three different cloud providers in under a year. We created a SQL interface to Kubernetes to hide the details of the underlying microservices implementation from the end user. We also developed our own reliable network protocol based on the Data Plane Development Kit (DPDK) for efficient data exchange between nodes in the public cloud. In this paper, we provide an overview of Yellowbrick and its microservices approach to delivering elasticity, scalability and separation of compute and storage. We also describe the optimizations we have implemented in the operating system and in our software to drive efficiency and performance, supported by benchmark results. We conclude with lessons learned and discuss future developments.

1 INTRODUCTION

The data warehousing industry has witnessed significant change over the past ten years. Prior to 2010, enterprise data warehouses were rooted in the private data center, focused on delivering efficiency and performance within a fixed resource footprint. As public cloud adoption has increased, bringing with it essentially limitless compute and storage resources, a new breed of cloud data warehouse has entered the market, characterized by elasticity, separation of compute and storage, and a SaaS user experience [1–3]. Over the same period, software development practices that combine container-based architectures with DevSecOps processes have grown in popularity, delivering software that is more scalable and resilient. Kubernetes [4] has become the *de facto* standard orchestration framework for containerized microservices. While several data warehouses advertise the ability to deploy on Kubernetes [5-7], none are composed from fine-grained microservices, and none provide a SQL interface to Kubernetes that abstracts configuration complexities from end users.

 14th Annual Conference on Innovative Data Systems Research (CIDR '24). January 14-17, 2024, Chaminade, USA.

2 OVERVIEW OF YELLOWBRICK

Yellowbrick is an ACID-compliant, MPP SQL relational data warehouse with a design centered on delivering instant elasticity, scalability, performance, efficiency, high concurrency, and availability. Yellowbrick consists of three major components: the *data warehouse manager*, providing the control plane for provisioning multiple, separate data warehouse instances; a *data warehouse instance*, which manages a set of databases; and elastically scalable *compute clusters* which add compute capacity to a data warehouse instance for different workloads (Figure 1).

In Yellowbrick, storage is separated from compute, and data is persisted in object storage as column-oriented, compressed files known as shards. Each compute node in a compute cluster has a locally attached NVMe (nonvolatile memory express) SSD-based shard cache to enhance query performance by caching shards read from object storage.

Each compute cluster can scale from 1 node to 64 nodes in single node increments. We refer to the MPP process running on each compute node within a cluster as the worker. Compute clusters can be configured to suspend and resume automatically based on incoming query activity, freeing up or provisioning the underlying cloud compute resources required in the process. All databases managed by an instance are visible to each compute cluster. Up to 3,000 workers can be attached to a single data warehouse instance, grouped into compute clusters. Users are assigned to one or more compute clusters, to which queries are submitted. In the case of assignment to multiple active compute clusters, an intelligent load balancer automatically routes to the one able to complete the query as fast as possible.

The data warehouse manager is used to provision data warehouse instances. It provides a web-based user interface from which data warehouses can be created, monitored and destroyed. It is designed to provide a single pane of glass from within which an administrator can provision data warehouses in the cloud of their choice as well as on Yellowbrick hardware running in a private data center.

Yellowbrick is designed to support multi-tenancy and runs within a customer's own cloud service provider account, enabling them to manage their own data and procure their own cloud computing and storage infrastructure. A common multi-tenancy use case sees data warehouse instances provisioned by the company's central IT department who act as a service provider to offer data warehousing as a service to different lines of business or even their own external customers.

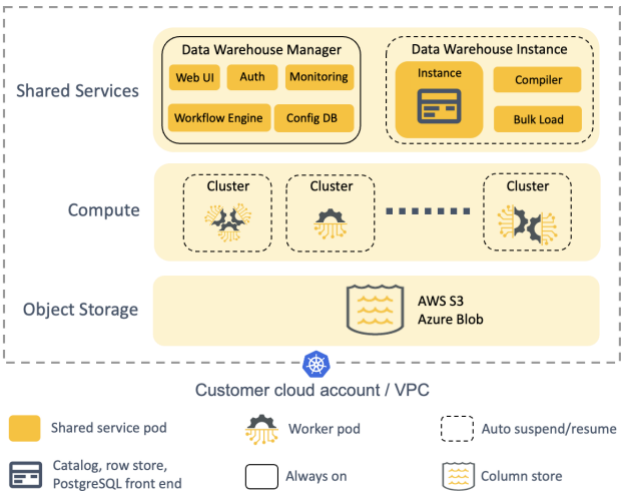

Figure 1: Yellowbrick architecture

2.1 Microservices Architecture

Yellowbrick is composed of a set of microservices that collectively deliver the database management system functionality. The microservices are packaged as Linux container images, and Kubernetes provides container orchestration and resilience, ensuring the data warehouse is maintained in the desired state. If a microservice crashes, Kubernetes automatically creates a replacement, even provisioning the cloud infrastructure needed to support it if necessary. The portability provided by Kubernetes and the underlying container runtime enabled us to port Yellowbrick to three different public clouds in under a year. The microservices that constitute Yellowbrick are depicted in Figure 2.

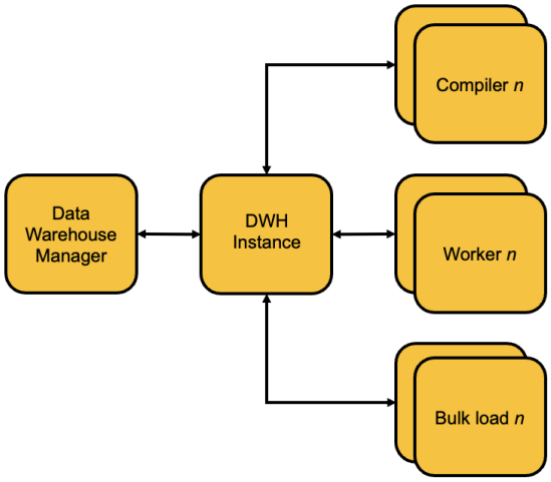

Figure 2: Yellowbrick microservices

The data warehouse instance is the front-end microservice for the data warehouse. This microservice manages connections to the data warehouse, as well as query parsing, query plan caching, row store, metadata management and transaction management duties, and is deployed as a singleton StatefulSet pod. Compute intensive tasks, such as bulk data loading and query compilation, are delegated to horizontally scalable ReplicaSet pods. The data warehouse manager consists of a set of pods providing UI, authentication, monitoring, configuration management and workflow services. The data warehouse manager supports one or more data warehouse instances.

Each compute node runs a single worker process deployed in a StatefulSet pod which is responsible for executing a portion of the query plan and managing the resources of the compute hardware. Each worker manages its own local storage which it uses as the shard file cache and for temporary spill space. New workers can be added or removed from a running compute cluster dynamically, and Kubernetes manages the process of spinning up new worker pods and new cloud hardware in response to changes to the cluster configuration.

An important design goal of Yellowbrick was to abstract the details of the underlying Kubernetes implementation from the end user. To that end, we built a SQL interface over Kubernetes to make the management of compute clusters straightforward. From the SQL command line, or from an ODBC/JDBC client, users can create, alter, suspend, resume, select, or destroy compute clusters.

For example, the directive to create a compute cluster follows the syntax:

```
CREATE [OR REPLACE] CLUSTER [IF NOT EXISTS]
<name>
WITH
     ( NODE_COUNT [=] <num>
     HARDWARE_INSTANCE [=] <name>
     WLM_PROFILE [=] <name>
     [ AUTO_SUSPEND [=] <num> | NULL ]
     [ AUTO_RESUME [=] TRUE | FALSE ]
     [ MAX_SPILL_PCT [=] <num> | NULL ]
     [ MAX_CACHE_PCT [=] <num> | NULL ] );
```

which creates a compute cluster with the specified number of workers, compute node type and workload management profile. The data warehouse instance parses the SQL directive and issues REST API calls to Kubernetes to provision the worker pods and compute nodes. We have also defined a number of system views whose base tables are populated directly by querying Kubernetes. These views provide information on cluster status, cluster events and configuration changes, including details of the user that made changes.

2.2 Deployment Approach

Another design goal was to make the process of deploying Yellowbrick as simple as possible. We wanted to provide an as-a-service-like user experience, even if the user is deploying the data warehouse in their own data center or in their own public cloud account. Ensuring that an administrator never has to see or touch a Helm chart was a hard requirement.

Yellowbrick deployments in AWS are bootstrapped using the AWS CloudFormation service. The service provisions a VPC, load balancers, subnets, security groups and an Elastic Kubernetes Service cluster with autoscaling enabled, and then starts the data warehouse manager services. Container images are retrieved from the Elastic Container Registry automatically. Once the underlying cloud infrastructure has been created, data warehouse instances can be created from the data warehouse manager UI.

At this point the workflow engine takes over behind the scenes to install a StatefulSet instance pod and ReplicaSet pods for the autoscaling compiler and bulk loader microservices in EKS. The required cloud compute instances are provisioned automatically by EKS autoscaling. Persistent volume claims are made to procure the necessary Elastic Block Storage volumes needed by the StatefulSet services. Then, via SQL or from the UI, compute clusters can be provisioned on EKS as described above.

Destroying compute clusters follows the same pattern. SQL command line actions or UI-driven directives are used to remove pods, triggering the release of underlying public cloud resources.

3 SOFTWARE OPTIMIZATIONS

We have implemented efficiencies throughout the database management software and have deployed a large number of innovative “OS bypass” techniques to work around inefficiencies in the Linux operating system in storage, networking, memory management and scheduling. We have also automated many of the tasks that are usually associated with managing and maintaining a data warehouse.

3.1 Database Optimizations

Yellowbrick’s query engine implements the standard SQL optimizations and algorithms one would expect of an enterprise MPP data warehouse, such as parallel query plans, cost-based optimization, workload management and parallel query execution. Query plans are translated to C++ code and then compiled by the compiler microservice and distributed to the workers for parallel execution. Yellowbrick is designed to handle ad hoc, batch and near real-time workloads of complex joins, aggregations, single record lookups, inserts, updates and deletes—simultaneously—over petabytes of data.

The SQL parser and planner are based on a fork of PostgreSQL 9.5. The query planner has been significantly modified compared to the original PostgreSQL planner, however the wire protocol and ODBC/JDBC drivers have been retained for reasons of ecosystem compatibility. Support is included for hash, sort-merge and loop joins as well as SQL rewrites for the pushdown, elimination, inference and simplification of predicates and joins. Cost estimation is used when planning joins, aggregates and scans. Primary and foreign key constraints declared in the database schema, while not used to enforce referential integrity in Yellowbrick, are used in join cardinality estimation along with statistics. Statistics are gathered and managed automatically using an implementation of the HyperLogLog algorithm [8].

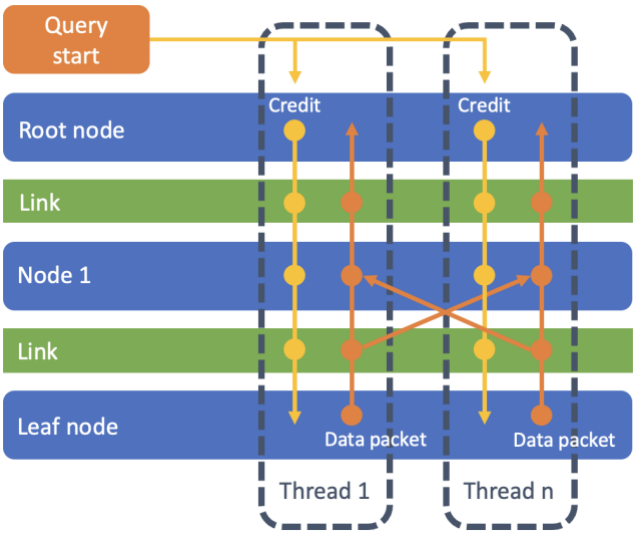

Figure 2: Query execution graph nodes are granted credits to process data. Credits flow downwards through the graph and data packets flow upwards

Yellowbrick is a shared nothing database, and MPP workers address a portion of the underlying data following one of three data distribution strategies. Rows are allocated to workers based on hash values in a specified column, randomly, or are replicated across workers. The type of distribution is specified on a per-table basis in the database schema. Depending on the query execution plan, data may be distributed between workers over the network.

Workers are comprised of an execution engine and a storage engine. The execution engine uses a credit-based flow control framework to govern the resources consumed by each query, constrained by the workload management rules that are in place. It also manages memory, threading, scheduling, communication with other workers, and the overall query lifecycle. The execution engine runs an object code instantiation of a query plan which is generated by LLVM inside the compilation microservice.

The execution engine processes a query graph whose nodes map to nodes in the abstract query plan generated by the SQL planner, as illustrated in Figure 2. The nodes in the query graph are the operators—such as table scan, join or sort—and the edges connecting the operators are links. The graph approach allows us to plan and execute complex query topologies, such as a table scan that feeds multiple consumers of data simultaneously.

At the start of query execution every thread on every worker is granted one credit. These credits are used to control the memory and temporary disk resources used by each query based on limits set by our workload management system. Credits flow down to the leaf nodes of the query graph and data packets flow up. Graph nodes can only process data packets if they possess a credit. Links in the graph manage connections between nodes; they account for credits and can distribute data packets to other threads, both synchronously and asynchronously. Leaf nodes are table scanning operators that read and filter data retrieved from storage.

A side effect of this credit-based approach to flow control is that the distribution operator, which moves data packets across the

physical network between MPP workers, also uses the same flow control and backpressure approaches—in essence, extending the query execution graph to be global across workers. The network buffers are the data packets themselves and they can be transmitted and received in place with no data copying when using RDMA, and with only one data copy on receive when using UDP with DPDK. Flow control guarantees optimal use of memory and keeps data cache-resident wherever possible.

Graph nodes execute in a cooperative fashion and cannot be interrupted against their will. Because multiple queries may be in progress on the execution engine on the same threads, graph nodes must explicitly yield control to enable the processing of other queries to proceed.

It is optimal to process data in different ways depending on the type of graph node in question. The execution engine supports row-oriented and column-oriented data packets. For example, the distribution node wants to operate in a row-oriented fashion because rows of data will be transmitted to different MPP workers depending on the hash of a column in the row. Likewise, the join node in the graph combines rows from two different tables and concatenates them. On the other hand, the table scanning node prefers to operate on columnar data straight from the storage engine, where it can take advantage of vectorized execution.

Graph nodes can choose the type of data packet format on which they are able to work, and transpose nodes are injected into the execution plan to optimally rearrange data accordingly. For example, data is transposed from columns to rows when moving between a table scan node and join node. Data is transposed to columns from rows when tabular data is written to disk.

The storage engine manages the column-oriented shard files. The table scanning leaf nodes in a query plan are executed by the storage engine. The query optimizer pushes filters down into the table scan, and the storage engine accepts conditions which will limit the rows and columns of the scan. These conditions are applied in the various phases of the scan to either skip a shard file entirely or skip components of it. Whole shards and parts of shards are skipped and filtered based on: filename, header and column metadata prior to decompression of actual values, and then after decompression with dynamically-created Bloom filters. In contrast to traditional sequential scans designed to optimize sequential disk reads, the storage engine drives millions of random IOPS to efficiently find only the data required.

The storage engine reads column-oriented shard data from the local NVMe cache over the PCIe bus, decompresses, transposes and filters it using vectorized SIMD instructions, and then passes packets up the query execution graph. Data packets are 256 KB in size and are designed to fit into L3 cache. The storage engine employs a custom NVMe driver that uses our own memory allocation scheme and runs in user space to avoid kernel overhead when accessing the local NVMe cache.

The execution engine is fully multi-core and NUMA-aware. Wherever possible, data packet processing is kept primarily core-local and secondarily NUMA-node-local; but in the event of skew, reallocation of packets across cores on a NUMA node will take place first, followed by reallocation across NUMA nodes, if necessary. This affinity of data packets and operators to cores and NUMA nodes also extends across the MPP network.

Figure 3: Query life cycle, illustrating the states that support restarting a query

Each compute cluster can adopt a different, configurable workload management profile. In our workload management implementation, compute, memory and temporary storage resources are split across pools. Rules map incoming queries to a particular pool based on attributes including user, role, application, database, query tag, and others. Queries can be assigned different priorities, throttled, and automatically cancelled and restarted within a different pool if they exceed given limits. Pools can be configured to allow mixed workloads (e.g. data loads and queries) to run on the same compute cluster without the need to manually partition workloads across different clusters. We have measured rates as high as 20,000 queries per second through our workload management system.

Figure 3 shows the life cycle of a query. Each query passes through several states while it is being prepared for execution, then it starts executing on the nodes in the compute cluster. Figure 3 identifies when queries can be cancelled or restarted by the workload management system based on the active rules. Once submitted, a query runs to completion, is cancelled, or fails with an error (DONE, CANCEL, and ERROR states). If a query is restarted or returns an error, it may re-enter the cycle in the ASSEMBLE state, but ultimately, all queries finish in one of the three completion states.

As a query passes through each state in its life cycle, runtime statistics are captured and logged. These statistics provide a measure of the time spent in each phase of query execution, giving administrators a means of monitoring and analyzing query performance. Wait times and actual processing times are measured at each stage.

3.2 Operating System Optimizations

For the sake of speed and efficiency, and to maximize time processing user data, Yellowbrick bypasses the Linux kernel for most system-level operations. The overall aim is to ensure that data read from NVMe SSDs is preserved in the CPU caches so that queries execute against data in the L3 cache rather than referencing main memory. To ensure this optimal data path is maintained we had to implement alternatives to the standard Linux memory management scheme and task scheduler.

At start up, our memory manager takes over control of the system memory to avoid kernel swapping. Memory allocations are grouped by query lifetime to avoid memory fragmentation. We have measured the performance of our memory allocator to be 100x faster than standard Linux in its implementation, and it is largely lock-free. The design is NUMA-aware and memory is pinned to specific NUMA nodes.

The memory allocator is initialized during initial setup of the C++ worker. All memory to be used by the allocator is mmaped in one contiguous virtual address region. The mmap request and subsequent analysis guarantee that the allocator only uses memory that is in either 2 MB or 1 GB HugePage blocks. The virtually contiguous HugePages need not be physically contiguous. The use of HugePages decreases the time required for the hardware to perform virtual-to-physical translation. These initial pages are mlocked, forcing them to remain in memory at their initial physical addresses. The memory allocator works entirely within this contiguous virtual address space. It leverages the contiguity to enable addressing with fewer bits, which saves space in the memory metadata storage.

We also implemented our own task scheduler that runs in user space and is 500x faster than the regular Linux task scheduler. Our implementation can context switch between queries in ~100 nanoseconds. The execution of a query is synchronized across a compute cluster so that every node is executing the same stage of the query plan at the same time. This helps to ensure that when data (re)distribution takes place, network queue depths do not build up to the extent that packets of rows end up in main memory instead of L3 cache.

Yellowbrick is a cooperative multitasking system. Time is divided into synchronized centisecond slots across a compute cluster. Only one query is processed across the cluster during this time slot and every CPU on every worker is entirely devoted to executing the current plan node for that query during the slot. At the end of the slot duration, the scheduler switches to process another query. The scheduler understands different query priorities, favors new work over longer running queries and coordinates across nodes in the cluster.

3.3 Networking Optimization using DPDK

Low latency, high bandwidth data exchange between worker nodes in the public cloud uses the Data Plane Development Kit (DPDK) [9] to bypass the kernel network stack, avoid intermediate copies and system calls, and directly address the network device from within user space. We developed a network protocol on top of UDP to provide reliable, ordered packet delivery and minimize CPU overhead. Our DPDK-based implementation provides a significant query performance enhancement compared to using the TCP/IP-based networking stack in Linux.

The use of DPDK within the database industry is not new. ScyllaDB [10], a distributed database compatible with Apache Cassandra, offers user-space networking through DPDK via the open-source C++ framework, Seastar [11]. While Seastar implements TCP in user space, it does not consider reliability over datagram protocols. The perceived difficulties in implementation and the need to recreate much of the networking stack that Linux already provides have impeded the application of DPDK in MPP database management systems to date.

In our implementation, DPDK is configured such that each vCPU thread on a worker connects to a corresponding vCPU thread on a different worker. Each thread has its own receive and transmit queues which are polled asynchronously. Receive side scaling is enabled to route packets between threads on a worker that share the same network interface card.

It is worth noting that, from our prior experiments, running DPDK inside a container does not impact performance versus running outside a container. A virtual function on a Single Root I/O Virtualization-enabled (SR-IOV) network interface card can be called directly from within a container. All major cloud providers offer access to such enhanced networking capabilities and allow multiple network interfaces to be attached to the same Kubernetes pod.

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

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

To illustrate the impact of DPDK on query performance, we executed a benchmark using the industry-standard TPC-DS [12] workload on Yellowbrick running in AWS. The Yellowbrick software can be configured to use either DPDK with our custom network protocol or TCP/IP. We performed a sequential run of the 99 SQL queries from the TPC-DS benchmark at the 1 TB scale. The benchmarking procedure followed [13] allows us to compare Yellowbrick performance with the published performance of other data warehouse platforms for this benchmark. In accordance with the benchmarking procedure, no tuning of the queries, the schema or the data was performed. From a data distribution perspective, rows are randomly allocated to the compute nodes rather than following a hash-based distribution strategy to maximize the volume of data exchanged between workers.

The total sequential runtime of the 99 TPC-DS queries is given in Table 1, utilizing TCP/IP and DPDK with our custom protocol for different compute cluster sizes. Over the entire sequential run, the compute nodes in the cluster exchange approximately 1.5 TB of data across the network.

The AWS EC2 instance type used in each cluster is the i4i.4xlarge, which provides a network bandwidth of up to 25 Gbps, a single 3.75 TB NVMe SSD drive and 128 GB of DRAM. DPDK utilizes two Elastic Network Adapters (ENA) on this node type, with 8 receive and transmit queues per ENA and each queue pair mapped to each of the 16 vCPUs.

From Table 1, the use of DPDK boosts the performance of this workload by almost 20%. The impact of the exchange of data between MPP workers during query execution is only one factor determining the runtime of a query. The runtime is also influenced by overall query complexity, the query plan chosen by the cost-based optimizer and storage I/O bandwidth.

Understanding the role of these other factors in query performance requires analysis of individual query runtimes. The improvement in individual query runtime as a function of the data exchanged between workers is illustrated in Figure 4 for a cluster with 4 workers.

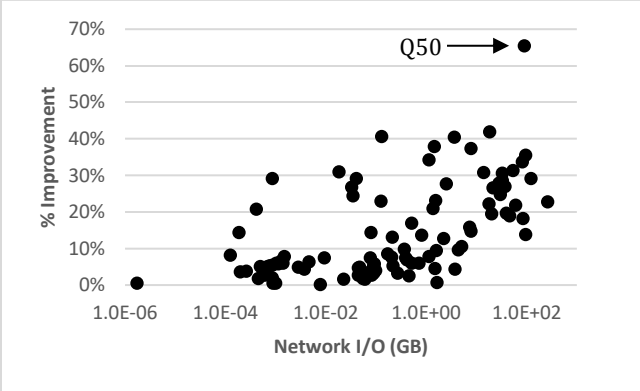

Figure 4: Runtime improvement of the 99 TPC-DS queries with DPDK vs TCP/IP as a function of data exchanged over the network for a 4-node compute cluster (semi-log x-axis)

The performance gains due to DPDK are correlated with volume of data exchanged between MPP nodes during the execution of a query, with the lowest gains associated with queries that exchange the least data. However, even queries that exchange a relatively modest amount of data can see significant performance increases due to kernel bypass and zero copy.

Cluster Size	Q50 Runtime with TCP/IP	Q50 Runtime with DPDK	Q50 Speedup with DPDK
2	74s	52s	29%
3	51s	15s	71%
4	30s	10s	66%

Table 2: Runtime of TPC-DS query 50 at the 1 TB scale as a function of compute cluster size and network implementation

The outlier, Q50, highlights the role query complexity plays in dictating the runtime improvement gained from DPDK (Table 2). Q50 demonstrated the largest improvement in execution time. The runtime for this query is dominated by the time taken to exchange rows over the network between workers during the hash join of two of the largest tables in the data set, containing 2.9 billion and 288 million records respectively. The nodes exchange ~100 GB during the execution of this query.

From Table 2, DPDK provides a 65-70% boost to the performance of Q50 for the two largest cluster sizes. The lower impact on performance in the case of the 2-node cluster can be attributed to network and local NVMe SSD storage bandwidth saturation.

3.4 Storage Optimizations

Most modern data warehouse implementations are backed by column stores [14]. While this approach can result in high data compression and good performance when querying a limited number of fields in a table, it is compromised in its ability to support efficient operations on single records. We opted for a hybrid storage engine design that combines a front-end row store and a back-end column store. The row store is managed by the data warehouse instance microservice.

From a query perspective, a table with data spanning both the row store and column store appears as a single logical table. Data can be inserted into the row store on a record-by-record basis at high speed and is instantly accessible. Rows are automatically flushed into the column store over time. Bulk loads of large amounts of data are inserted directly into the column store via parallel connections to the workers, bypassing the row store.

ACID properties are preserved across the row and column store by using a common transaction log with a “read committed” level of isolation and multi-version concurrency control. Shard files are immutable, and deleted records are tracked through the presence of side files containing bitmaps that mask the deleted rows in their respective shard. Shard files and deletion files are merged periodically to create new shard files.

Workers read data in 256 KB blocks from the object store and cache them locally on NVMe SSDs using a mixed strategy of single block reads and prefetching. This read block size provided reasonable tradeoffs between read IOPS, throughput and NVMe cache efficiency in our experiments with AWS S3.

A variant of the standard LRU policy providing basic scan resistance (newly inserted pages are placed further down the list, and only promoted to the head on second access) governs NVMe cache eviction. In the case of data loading, records are persisted directly in object storage while workers notify each other of changes to shard file ownership that will affect their caches. Following compute cluster resize, shard file ownership amongst the workers is reallocated using Rendevous hashing [15]. During query execution, data is read from the NVMe cache in blocks of 32 KB or less across the PCIe lanes into L3 CPU cache.

Shard files are ~100 MB in size and are transactionally written to the object store in 2 MB blocks. Each block is written with a single PUT operation. Data committed to Yellowbrick is written around the NVMe cache and into object storage; the cache is only populated through read operations. Experiments performed on AWS S3 indicate 2 MB writes provide the optimal bandwidth and throughput for our use case.

We implemented our own C++ S3 connection library [16] to support deployments on AWS. The connection library is used by

the workers to read and write shard files to S3 buckets. We found the standard AWS C++ S3 client library to be somewhat inefficient. Our library delivers 3x the throughput, saturates the network, and uses a fraction of the CPU compared to the AWS implementation. Performance and efficiency gains were realized by greatly reducing the number of data copies and memory allocations, pipelining HTTP/HTTPS requests, and through prudent socket management.

4 PERFORMANCE COMPARISON

In Table 3, we compare our TPC-DS 1 TB timings and costs when deployed on AWS with results published for other data warehouse platforms [13]. The configuration of each platform was selected based on a broadly similar hourly cost [13].

Platform	Configuration	Total Runtime	Relative Runtime	Cost per hour
Yellowbrick	4x i4i.4xlarge	995s	1	$8.42
Snowflake	Medium	2690s	2.7	$8.00
Redshift	3x ra3.4xlarge	3199s	3.2	$9.78
BigQuery	300 Slots	2298s	2.3	$8.22
Synapse	DW500c	4846s	4.9	$6.00
Databricks	4x i3.2xlarge	2974s	3.0	$7.22

Table 3: Comparison of the sequential total runtimes of the 99 TPC-DS queries at the 1 TB scale across competing data warehouse platforms

As in Section 3.3, we followed the same methodology when executing the benchmark on Yellowbrick, first warming the NVMe SSD caches on each compute node by running a full table scan for each table in the schema, and then timing one sequential run through the 99 SQL queries in the benchmark. No modifications to the published [13] TPC-DS queries or schema were made. For these configurations, a 4-node Yellowbrick cluster executes the TPC-DS 1 TB workload 2-5x faster than the other platforms.

The relative cost per query is an important normalizing metric since it accounts for both the price and the performance of each platform. The optimizations described in the previous sections result in cost savings of 2-3x over the other platforms based this synthetic benchmark (Figure 5).

An interesting, but not surprising, feature of Figure 5 is how similar in price-performance terms the other data warehouse platforms are to each other, excluding Yellowbrick. After all, most modern data warehouse platforms incorporate the same standard SQL and database software optimization techniques, such as automated SQL rewrites, cost-based optimization, caching, SIMD operations, columnar storage and zone maps. They are also running on similar public cloud hardware. To differentiate in the market in terms of price-performance, we believe that a data warehouse platform must optimize in the Linux kernel to tailor low-level operations to meet the requirements of data warehouse workloads.

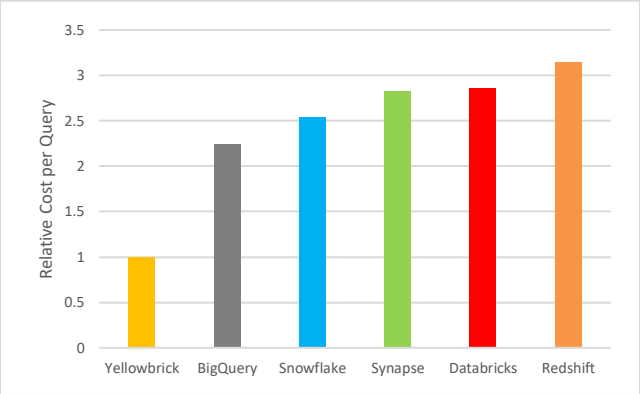

Figure 5: Relative cost per query across the platforms listed in Table 3 for the TPC-DS benchmark at 1 TB scale

4.1 Scaling Compute and Data

We characterized the scaling capabilities of Yellowbrick by performing sequential runs of the TPC-DS queries against different scale factors and compute cluster sizes (Figure 6). The total runtime remains approximately the same for a given ratio of data to node count as the data volume increases from 3 TB to 100 TB.

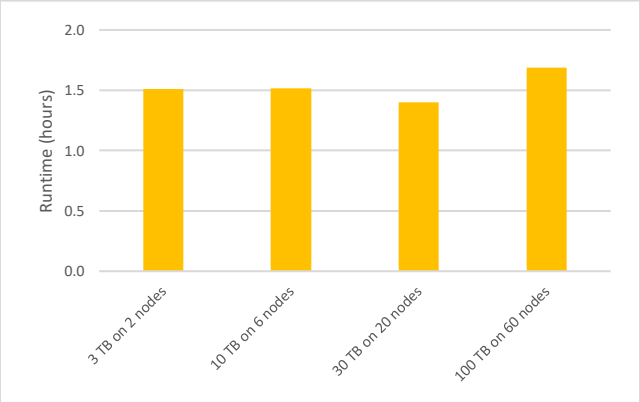

Figure 6: Runtime scaling as a function of compute cluster size and data set size (3 TB to 100 TB) for the TPC-DS benchmark

Data set	Configuration	Total Runtime	Cost per TB per run
3 TB	2x i4i.4xlarge	5430s	$2.72
10 TB	6x i4i.4xlarge	5451s	$2.04
30 TB	20x i4i.4xlarge	5036s	$1.94
100 TB	60x i4i.4xlarge	6071s	$2.06

Table 4: Price-performance per TB per sequential run of the TPC-DS queries over varying data volumes and compute cluster sizes

The linear scaling of price-performance is shown in Table 4. The results demonstrate that Yellowbrick will scale predictably from a price-performance perspective as data volumes and workloads grow.

4.2 Concurrency Scaling

We also performed measurements of the concurrency scaling characteristics of Yellowbrick. For these tests we used the 1 TB TPC-DS data set and workload. Each compute cluster has a maximum concurrency limit of 100 queries. Workloads that exceed this limit are queued. The concurrency limit can be configured through a workload management profile and so trade-offs can be made between the degree of concurrency and the memory and temporary spill space available to each query on a per cluster basis.

We configured a 20-node cluster of i4i.4xlarge EC2 instances with 64 concurrency lanes and ran 1, 2, 4, 8, 16, 32 and 64 parallel streams of the 99 TPC-DS queries in turn. Each stream executed a random sequence of the 99 queries.

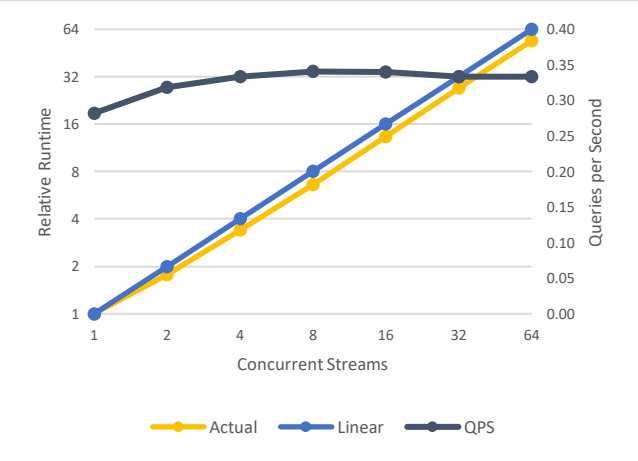

Figure 7: Relative TPC-DS workload runtime and query throughput for a varying number of concurrent streams using a 20-node i4i.4xlarge cluster with data at the 1 TB scale factor

Yellowbrick's workload management scheme does not reserve CPU capacity for each lane a query runs in. If only one query is running on a cluster then the query is allocated all the available CPU in the cluster. As more queries are submitted, CPU is allocated to queries depending on their relative priority which can be set through workload management rules. In the case of equal priority, one would therefore expect the time taken to execute the same query concurrently in two lanes would be double the runtime of the query running on its own.

Figure 7 shows the runtime relative to a sequential run of the 99 TPC-DS queries for different degrees of concurrency on a 20-node cluster. As expected, the runtime doubles as the degree of concurrency doubles. Yellowbrick's actual concurrency scaling is better than linear for this workload, likely due to data sharing between concurrent queries in different lanes accessing data already present in the CPU caches. Note also that the query throughput remains consistent across increasing degrees of concurrency, again ensuring predictable price-performance as workload volumes increase.

We extended the testing to examine the impact on runtime and query throughput with different cluster sizes while maintaining the degree of concurrency at 64 streams (Table 5).

For 64 concurrent query streams, increasing the size of the cluster decreases the runtime and increases the query throughput, albeit with diminishing returns for this level of concurrency and workload.

Configuration	Query Streams	Query Count	Total Runtime	Relative Runtime	QPS
10xi4i.4xlarge	64	6336	29174s	1.0	0.22
20xi4i.4xlarge	64	6336	19011s	0.7	0.33
30xi4i.4xlarge	64	6336	16110s	0.6	0.39

Table 5: TPC-DS workload runtime and query throughput for 64 concurrent streams for a 10, 20 and 30-node i4i.4xlarge cluster with data at the 1 TB scale factor

4.3 Multi-cluster Scaling

The concurrency and the scaling characteristics for a single compute cluster depend on the complexity and resource requirements of the workload. To avoid queuing queries on a single cluster, and to scale query concurrency linearly, our customers can take advantage of Yellowbrick's in-built query load balancer to distribute queries across more than one cluster. The load balancer allocates in-bound queries across clusters on a least-busy basis.

We extended our scaling tests from Section 4.2 to demonstrate the additional query concurrency and throughput that can be obtained by distributing workloads automatically over more than one compute cluster using the load balancing mechanism. In this experiment, we added a second and third cluster of 20-nodes alongside the original 20-node cluster, increasing the concurrency level to 128 streams and then to 196 streams of randomly ordered TPC-DS queries.

Cluster Count	Query Streams	Query Count	Total Runtime	Relative Runtime	QPS
1	64	6336	19011s	1.0	0.33
2	128	12672	19116s	1.0	0.66
3	192	19008	18949s	1.0	1.00

Table 6: TPC-DS workload runtime and query throughput for 64, 128 and 192 concurrent streams using load-balanced multi-cluster configurations (20 compute nodes per cluster) with data at the 1 TB scale factor

Table 6 shows that when the workload is shared across multiple compute clusters, the query throughput increases linearly as the degree of query concurrency increases linearly.

A common pattern our customers follow is to define a single cluster configuration that satisfies the performance requirements for a certain number of users, and then add additional clusters following this sizing template as their user community and business grows. The load balancer ensures that applications are oblivious to multi-cluster expansions. Our customers can start small with as little as a one node compute cluster, expand this cluster node-by-node, and then add more clusters as needed.

5 CONCLUSIONS AND FURTHER WORK

Yellowbrick's adoption of Kubernetes as the orchestration and platform-agnostic runtime enables it to deliver a modern data warehouse that runs anywhere. Kubernetes does the heavy lifting when it comes to managing the lifecycle of the data warehouse, providing elasticity, availability, and scalability. The delegation of infrastructure responsibility to Kubernetes has allowed us to focus on the core business of enhancing database performance and adding new features. Work to optimize our database software, the network protocols and in the kernel has not been impeded by Kubernetes. We are still able to efficiently access low level devices such as NVMe SSDs and network interface cards even in virtualized public cloud environments.

The optimizations implemented to reduce OS kernel overhead in Yellowbrick contribute significant performance benefits. As our benchmarks show, using a custom network protocol based on DPDK for the exchange of data between MPP nodes alone reduces the runtime of some queries by as much as 70% in the public cloud. We also demonstrated how Yellowbrick scales linearly along dimensions including: compute cluster size, number of compute clusters, data volume, and degree of query concurrency. The implication of these results is that the price-performance of Yellowbrick can be reliably predicted, and customers can be confident that Yellowbrick will scale as their business grows.

As a next step, we are investigating the impact of compression on networking performance. Initial results indicate a 50% reduction in network data volume with lz4 compression. We are also evaluating the performance impact of offloading the compression overhead to the network interface card.

Additional performance improvements in progress include: extending our range-based filtering to support multiple ranges per column; support for more complex Bloom filter expressions and the selective application of filters based on cost; and further automated SQL query rewrites in the planner. In our testing, manual tuning of the TPC-DS queries results in a further 2x speedup for this workload on Yellowbrick. However, tuning our query planner to perform well against this artificial workload is low on our priority list.

REFERENCES

[1] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. 2016. The Snowfake Elastic Data Warehouse. In SIGMOD. San Francisco, CA, USA, 215–226, 2016.

[2] BigQuery. https://cloud.google.com/bigquery/docs/introduction. Accessed: May 30, 2023.

[3] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In SIGMOD. Melbourne, Victoria, Australia, 1917–1923, 2015.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. 2016. Borg, Omega, and Kubernetes. Queue, 14(1), 2016.

[5] Altinity ClickHouse Operator. https://github.com/Altinity/clickhouse-operator. Accessed: May 30 2023.

[6] O. Basarir, L. Hamel, J. Patel, D. Sharp, G. Tadi, F. Yang and X. Zhang. 2019. Pivotal Greenplum for Kubernetes: Demonstration of Managing Greenplum Database on Kubernetes. In SIGMOD. New York, NY, USA, 1969–1972, 2019.

[7] Vertica on Kubernetes. https://www.vertica.com/blog/vertica-on-kubernetes/. Accessed May 30, 2023.

[8] Flajolet, Philippe; Fusy, Éric; Gandouet, Olivier; Meunier, Frédéric. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. Discrete Mathematics and Theoretical Computer Science Proceedings. Nancy, France. 137–156, 2007.

[9] Linux Foundation. DPDK. https://www.dpdk.org/. Accessed: May 30, 2023.

[10] ScyllaDB Inc. ScyllaDB. https://www.scylladb.com/. Accessed: May 30, 2023.

[11] Seastar. http://seastar.io/. Accessed: May 30, 2023.

[12] TPC-DS. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf. Accessed: May 30, 2023.

[13] Fivetran Cloud Data Warehouse Benchmark. https://www.fivetran.com/blog/warehouse-benchmark. Accessed: May 30, 2023.

[14] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In VLDB 2005. 553–564, 2005.

[15] Thaler, David and Chinya Ravishankar. 1996. A Name-Based Mapping Scheme for Rendezvous. University of Michigan Technical Report CSE-TR-316-96, 1996.

[16] N. Carson. Improving S3 performance from C++. https://www.neilcarson.me/s3iops/. Accessed May 30, 2023.