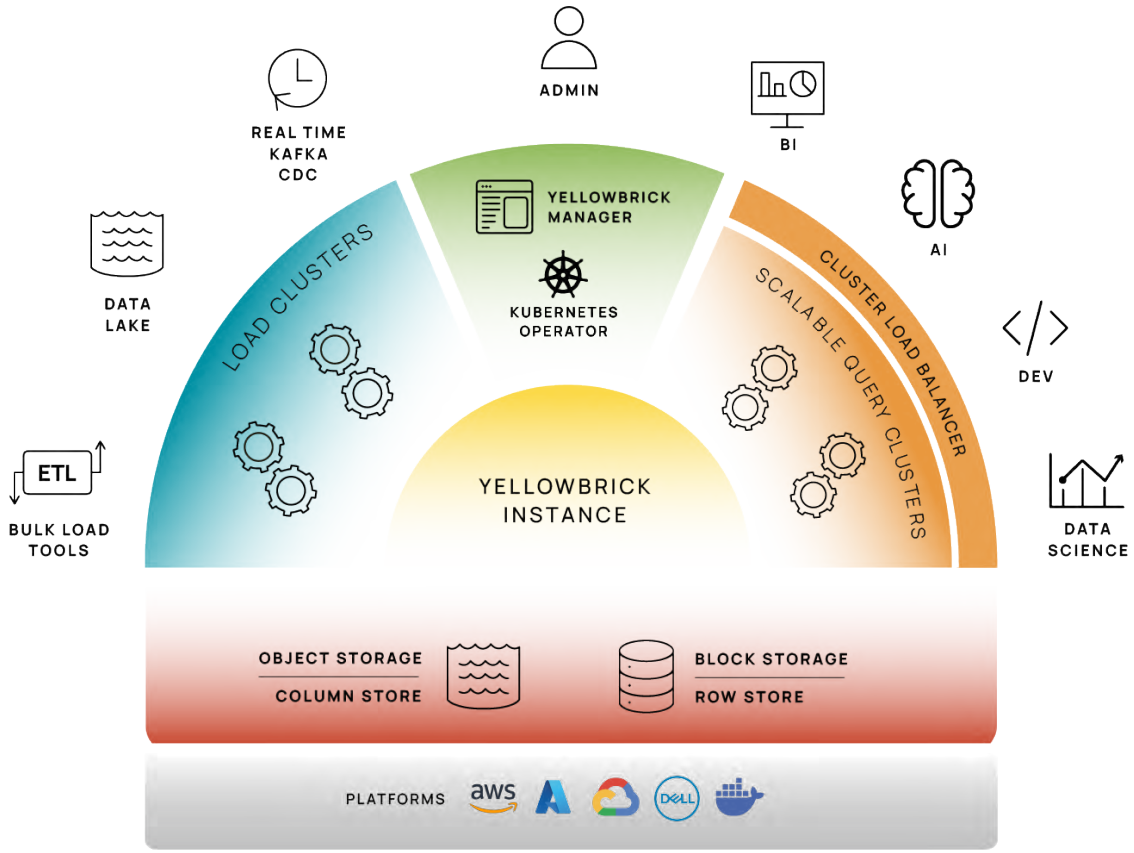
Yellowbrick

WHITEPAPER

Inside the Yellowbrick Data Platform



Contents

Introduction

The Yellowbrick Data Platform is a cloud native, parallel SQL database designed for the most demanding batch, ad hoc, real-time and mixed workloads. Fully elastic clusters with separate storage and compute run complex queries at multi-petabyte scale with sub-second response times.

Yellowbrick innovates in three key areas:

- **Security:** Data is protected, compliance standards are met, and organizations retain complete control.
- **Performance:** Unparalleled speed and scalability while efficiently consuming resources to maximize return on investment.
- **Simplicity:** SaaS-like management experience, familiar SQL RDBMS, no optimization required, and running Kubernetes for unified operations across any environment.

This document explains the architectural approaches that support these innovations. They cross everything from the creative usage of Kubernetes to cluster management, to data storage, to query planning and execution, and even the user interface itself. Yellowbrick is designed to run Tier 1 enterprise-grade data platform with the following characteristics:

- **Modern, elastic architecture:** Elasticity, separate storage & compute, driven completely through SQL.
- **Run in the customer's cloud account:** Yellowbrick customers pay for their own cloud infrastructure, make use of their own cloud storage and control their own data security. Yellowbrick doesn't see or store user data or queries.
- **Run across all public clouds:** At the time of writing, Yellowbrick supports AWS, Azure, and GCP public clouds.
- **Run on-premises:** Provide the same elastic user experience as in the cloud on-premises.
- **Hardened SQL support:** Yellowbrick can reliably execute complex ANSI-standard SQL queries across massive schemas, supporting complex join hierarchies, correlated subqueries, deeply nested CTEs, stored procedures, etc.
- **Execute complex workloads:** Yellowbrick handles highly concurrent mixed workloads with continual bulk and real-time ingest, merging, and highly concurrent queries with guaranteed quality of service through workload management. Full ACID transaction semantics are present throughout the stack.
- **Reliability:** Yellowbrick is highly available and resilient to node, storage, and network failure. Workloads across compute clusters are isolated from one another.
- **Support for disaster recovery:** Asynchronous replication of both data and DDL with read-only hot standby instances is built in.
- **Support for data retention and business continuity:** A mature enterprise-level backup scheme for off-site data retention supports incremental, cumulative, and full backups and object-level restore.
- **Mature ecosystem:** Yellowbrick has enterprise support agreements with all major BI, ETL, data mining, CDC, and machine learning vendors.
- **Best query efficiency in the industry:** Yellowbrick executes ad-hoc queries against large data sets incredibly efficiently, making use of many technical innovations in query execution.
- **Flexible pricing and consumption:** Support for subscription contracts as well as workload management allows variable numbers of concurrent users with a predictable price.
- **Open interfaces:** By making use of PostgreSQL's wire protocols, *DBC drivers, and metadata schema, Yellowbrick is comfortable for developers and DBAs to work with. Open-source integrations for tools like Kafka and Spark are standard.

Yellowbrick isn't a SQL-on-Hadoop type of product, or a solo query engine running on top of other open source infrastructure. It's a database that organizations can trust to store and be the system of record for their most valuable enterprise data, and to generate business-critical, auditable financial reports that their businesses depend on. It requires almost no management, tuning, diagnosing, or handholding and is familiar to modern developers accustomed to working with PostgreSQL.

Cloud Native Cluster Management

Full elasticity with separate storage and compute is now a table stakes feature for any modern data platform. BigQuery and Snowflake set the bar here, the latter allowing explicit control of clusters ("virtual warehouses") within the database through an expressive SQL grammar. Both technologies were conceived before Kubernetes was invented. Kubernetes (K8s) has been a double-edged sword: On one hand, it provides a declarative, standard, and portable interface for running scale-out applications across many different cloud vendors and stacks (aka "the new cluster operating system"); but on the other, it places a burdensome and daunting management overhead onto the teams that run such clusters due to its extreme complexity.

At Yellowbrick, Kubernetes is used internally to gain portability across cloud platforms without requiring end users to know anything about it. Users never touch or see Kubernetes, a helm chart, pod, node, operator, or configuration file and aren't aware they even exist. The user experience largely mirrors that of Snowflake: creating, growing, and destroying compute clusters through SQL utility statements, and viewing the state of the scale-out data warehouses through SQL tables. Users simply provision a warehouse and create an elastic cluster through one line of SQL (or a few clicks of the UI) and off they go.

Yellowbrick Instance Topology

Each instance is a fully independent database supporting multiple elastic compute clusters with separate storage and compute. Data warehouse instances are independent of one another but can be managed through a single pane of glass with the Yellowbrick Manager. There is no coupling of availability between instances, and an error in one instance cannot affect another instance. Each instance is deployed into one Kubernetes namespace. Figure 1 shows the components that make up each instance:

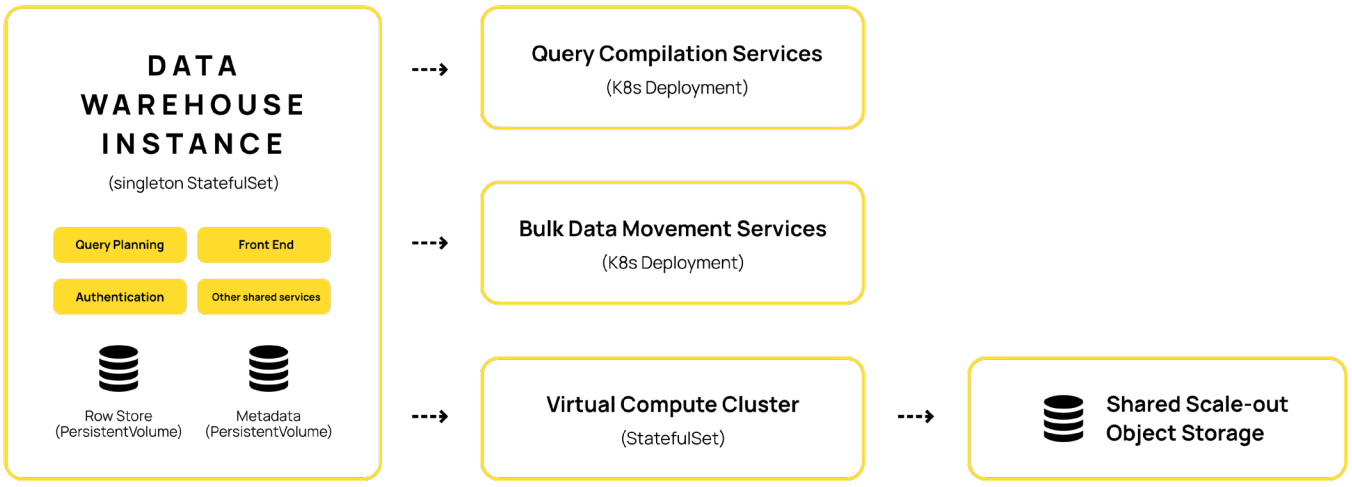


FIGURE 1

The instance pod is the front end of the database. It provides shared services such as connection management, metadata schema, and query planning. The entire instance can be suspended and resumed. Yellowbrick's row-oriented storage for real-time ingest is also located in the instance pod to provide the lowest possible commit latency. No query execution occurs on the instance pod; query execution is deferred to elastic compute clusters.

Virtual Compute Clusters are required to write data and execute SQL queries. Each compute cluster operates independently of one another. SQL utility statements support creating clusters, assigning users to clusters, suspending, resuming, and resizing. Since multiple compute clusters work on the same shared data, which is cached locally and persisted in object storage, different clusters can be used for different purposes. For example, spinning up a cluster on demand to perform ETL, having a different cluster for business-critical reports vs. ad-hoc queries, or simply creating more clusters to support higher levels of concurrency.

Query compilation services turn SQL statements into vectorized, executable code. Query plans and compiled code are aggressively cached for re-use. Query compilation services pods scale up and down automatically based on CPU consumption.

Bulk data movement services support bulk loading and unloading of data and scale out automatically, depending on the number of concurrent requests. System tables are provided for querying the current state of all the above services. For example, sys.cluster shows the current state of all elastic compute clusters.

Implementation Using Kubernetes

SQL-based management of elastic compute clusters is implemented using Kubernetes while hiding details from the users and DBAs. When SQL commands to scale compute clusters or supporting services are issued, the Instance pod uses the Kubernetes API to create or alter underlying API objects (for example, pods, namespaces, configmaps, and events) and monitor progress in affecting the changes. Similarly, when querying system views to examine the state of clusters, the Instance pod executes Kubernetes API requests to list the relevant objects, join the Kubernetes API objects with locally stored metadata, and return the view to the user.

The very existence of Kubernetes is never exposed to users, but by leveraging this containerized, cloud-native architecture, Yellowbrick gains trivial portability across all the public clouds and modern on-premises infrastructure. This "SQL user interface for Kubernetes" is a Yellowbrick innovation that is not available in other data platforms; most don't support the management of elasticity and scaling through SQL, and those that do aren't implemented on top of Kubernetes. It also becomes possible to embed and scale Yellowbrick inside other microservices-based, cloud-native applications, even to the point of automating instance creation and provisioning through provided GraphQL APIs.

Tracking Consumption and Auditing Changes

All changes in the level of billable resources are captured in system tables. The core unit of consumption in Yellowbrick is the vCPU/second of cluster compute resources used; this is aggregated hourly and daily and can be queried through system tables, summarizing by user or by cluster, etc.

Direct Data Accelerator® Architecture

The algorithms used by most databases built in the last two decades have several consistent core assumptions:

- Databases executors should cache data in a buffer cache in memory to avoid reading too much data from disk.
- Core operators, such as joins, sorts, and aggregates, should use a buffer pool to transparently support spilling data to disk in case memory capacity is exceeded.
- Keeping more data "in memory" is the way to better performance – after all, CPUs can copy memory far faster than they can move it from the storage or network.
- Tight integration with Linux provides performant storage and network IO and threading.

Yellowbrick turns most of these assumptions on their heads because on modern servers available in the cloud and on-premises, it's now the case that:

- The maximum effective bandwidth of reading data from high performance, direct attached storage can be the same as from main memory.
- Moving data across the network can be less resource-intensive than copying it from/to main memory.
- The CPU can run 10x faster if processing data from caches, rather than from main memory.
- Linux simply can't scale IO effectively at the levels of concurrency offered by today's servers.

These advances in core computer architecture have allowed us to build the Yellowbrick database engine in a manner that supports much more efficient query execution than other databases. It's not uncommon to find Yellowbrick using 1/4 or less vCPU resources per query than competitors, resulting in substantial cost savings for customers.

To take advantage of the latest technology, Yellowbrick developed a software data path that's much more efficient than that available in Linux. Linux's limitations are quite widely known in the network security and low-latency stock trading community, where the most advanced software developers will talk about using OS bypass technologies such as DPDK or OpenOnload with Linux hugepages to fix such problems. However, these leading-edge technologies from Intel and others are far too restrictive to be sufficiently general purpose for running a parallel database engine.

Yellowbrick's own OS bypass technology is specifically designed for analytic database engines and – due to recent advances in Kubernetes and container architecture – can be used in any standard container environment.
Yellowbrick has developed:

- A networking stack for high-speed data exchange between parallel processing nodes that's up to 20x more efficient than built-in Linux.
- A storage I/O stack and file system with 1/100 of the cost per IO compared to standard Linux.
- An S3-style object storage stack that supports all three cloud providers at 1/10 of the CPU usage compared to the vendors' own software.
- A process and thread management subsystem with no measurable overhead.
- A cluster-based scheduler to allow synchronized query execution across cluster nodes.
- A memory management approach optimized for the above.

We'll discuss some of these innovations below.

The Need to Optimize for Modern Computer Architecture

Today's servers are routinely available with over a terabyte of memory and over 100 CPU cores, and the trend of more cores and memory is set to continue. Running generic software on these instances does not work well because operating system schedulers were built to wait for events and "context switch." Threads wait for events – such as a keypress, a network packet arriving, a storage I/O completing or synchronization primitives becoming available – and switch between competing threads and processes to try to be as fair as possible and use buffers efficiently. As a result, it's not uncommon for modern databases to do tens of thousands of context switches per second per CPU core, and millions of them per second in aggregate.

Conventional wisdom states that if you're not spending much CPU time context switching – under 10% – you're in good shape; context switches are cheap with a good operating system. However, this assumption is outdated. Modern CPUs get their performance from processing data from their caches, typically called L1, L2, and L3. The L1 cache contains data pertinent to the most recent processing, the L2 cache is larger but slower to access, and likewise the L3 cache. The L1 cache per CPU core is measured in tens of kilobytes, the L2 cache in hundreds of kilobytes, and the L3 cache in single-digit megabytes. If you were to look at the speed at which you can access data on a modern CPU, you'd see ratios like those shown in Table 1.

Memory Type	Access latency (in nanoseconds)
L1 cache	1.5
L2 cache	5.3
L3 cache	24.5
Main memory	120

TABLE 1

When the CPU context-switches inside a database, the different contexts may be doing tasks such as:

- Running a hash join of two tables for Tim.
- Sorting data for Torben.
- Running a hash join of two tables for Marc.
- Waiting to look up a block in the filesystem for Mark to scan a table.
- Preparing data to send over the network in TCP packets for Jason.

Each of these tasks is accessing its own large data structures and dealing with its own cached data. Each context switch requires moving new data in and out of the CPU caches, invalidating data that was there before.
One might ask, "Well, what if we were to be able to do these tasks sequentially: Finish Tim's join, then Marc's join, then

Torben's sort, then Mark's work, then Jason's? Doing things one at a time, we'd do less context switching and use our caches more efficiently, right?" This is partly true, but it doesn't end there: The minute Joe's join needs to send data over the network, the CPU enters the Linux networking stack – around 100,000 lines of complex code that will do a great job changing memory mappings, filling the caches, and evicting Joe's hash tables, only to force them to be reloaded from main memory when the network processing is done. Similarly, if Joe's join needs to read a new disk block, it will enter the Linux I/O stack and filesystem, and hundreds of thousands of lines of code and complex data structures will also handily displace his hash tables.

When this context switching and bouncing in and out of complex Linux kernel subsystems is happening continuously across dozens of cores, any modern CPU will struggle to work efficiently. The DBAs will be none the wiser because the CPU will be 100% utilized, but under the covers, the database is achieving only a fraction of its theoretical maximum efficiency. Therefore, most data platforms today support very low levels of concurrency.

Design of the Direct Data Accelerator®

Yellowbrick's Direct Data Accelerator® implements a new execution model to work around the issues above by eliminating measurable context switching overhead and penalties associated with accessing storage, the network, and other hardware devices. This is done with a new, reactive programming model for the entire data path. Some of the principles of this new programming model are described below.

Threading, Processes, and Scheduling

Yellowbrick has a new threading model based on reactive concepts such as futures and coroutines. Small, individual units of work called tasks are scheduled and run to completion without preemptive context switching. Tasks do not have stacks associated with them. Tasks in Yellowbrick never block (as far as the OS is concerned) and no stacks are preserved when waiting for futures; instead, it's up to the caller to optimally preserve state in either per-thread data structures or lambdas. Although tough for general-purpose programming, in a constrained environment such as a database, this model is tractable and offers massive efficiency improvements.

Yellowbrick implements synchronization primitives as well as parallel iterators to protect access to shared resources. Awareness of multiple CPU cores and NUMA nodes is intrinsic to the system from top to bottom: The author of code, written in assembly or C++, makes deliberate decisions as to the parts of the compute complex on which to run. Optimal memory allocations, from the closest or most recently cached data structures, will be provided automatically to the runtime, and even the handling of bizarre modern CPU artifacts (such as cache aliasing of stacks) is built in.

In a traditional OS, a process comprises threads that execute. Yellowbrick implements a different type of process model: A work comprises tasks that execute in a fully asynchronous, reactive manner. Works have their own memory arenas from which to allocate, which are torn down together, and all resource consumption of the work is bounded and isolated by the kernel: what fraction of CPU it can use, how much memory, how much compute, how much disk storage, and so on.

The scheduler is aware of works and tasks and will try not to intermix the execution of tasks from different works to further

avoid cache displacement. It even goes one step further and tries to ensure that, when database queries are exchanging large amounts of data (such as re-distributing data for a large join), the same work is running on peers in the cluster at exactly the same time, such that received data may be processed immediately. Multi-tasking is cooperative, rather than preemptive.

Works are created programmatically from built-in code, or dynamically loaded and unloaded at runtime, just like processes in Linux. The latter approach is used by the database for runtime query loading and unloading.

Device Access

The database needs to access network devices, storage devices, and hardware accelerators when available. Traditional device drivers run in the Linux kernel and interrupt execution whenever something happens. In contrast, all Yellowbrick device drivers are asynchronous and polling in nature. Access to drivers is always via queue pairs – command and completion queues – with well-defined interfaces. Drivers are present for general PCIe devices, NVMe SSDs, various network adapters, and so on, all of which work without Linux's involvement. In cases where Yellowbrick is running without bypass being available, emulated drivers for each class (network, storage, etc.) are present that fall back on the Linux kernel or software emulation.

Local File System

Yellowbrick implemented a local file system called BBFS (Big Block File System), that builds on top of the device access layer to provide a full namespace on top of raw storage devices. It provides most expected file system semantics – directory management, creating files, open, close, read/write, readv/writev, delete, rename, etc. – but is implemented completely asynchronously.
Metadata and metadata locks are heavily sharded to maximize concurrency. The file system metadata itself is tiny, fitting in indexed in-memory structures for fast lookups. A read() operation on a file can be submitted through the entire file switch, file system, and device drivers in only a few hundred CPU cycles.

Network Data Exchange

Like many modern microservices-based software stacks, Yellowbrick is implemented in a variety of different languages: Primarily C, C++, and Java, with a sprinkling of Go and Python where necessary. These services need to talk to each other. For maximum efficiency, Yellowbrick contains a highly efficient communication framework called YBRPC that's optimized for the latest server instances. Several different underlying YBRPC transports are in use:

- YRD (Yellowbrick Reliable Datagram): This is built using Intel's DPDK library for OS bypass. It's a datagram-orient protocol that implements checksums and re-transmissions as needed to guarantee reliability and be able to route across sub-nets. Due to wide support for DPDK, we use this protocol on all cloud-based instances. YRD enables zero-copy sends and single-copy receives with a fraction of the CPU overhead of TCP.
- RDMA (Remote Direct Memory Access) is a networking technology used to move data directly between the memory of servers as efficiently as possible. It works over both Ethernet (where supported) and InfiniBand fabrics, primarily for

on-premises instances. Sends and receives are both zero-copy and latency is measured in nanoseconds.

- Linux TCP: A transport layer for generic kernel-based TCP which is used when other more efficient transports are not available.
- Unix Domain Socket: When processes are known to execute on the same node, this is more efficient than using a TCP socket.

The custom network stacks pay substantial dividends in query execution efficiency: Yellowbrick benchmarks have clocked a single CPU core sending and receiving 16GB/sec of data across the network in the MPP fast path, with time to spare. When using the Linux kernel, around 1.5GB/ sec is the limit and the CPU core is fully loaded, leaving no time whatsoever for data processing. YBRPC allows expensive parts of database queries – such as re-distribution of data for joins, aggregates (GROUP BY), and sorting – to run 10x more efficiently than competing databases, using a fraction of the resources.

Object Store Access

In cloud architecture, local storage is ephemeral, so the only way to reliably persist data is by writing it to some form of remote storage, and Amazon S3/Azure ADLS/GCS object storage is the most cost effective. Remote storage has issues with latency, though, so to maximize bandwidth and IOPS, large IO queue depths across many targets must be correctly pipelined. The client libraries from the cloud vendors are incompatible and all third-party libraries are incredibly inefficient, performing gratuitous data copying and dealing poorly with pipelining the massive numbers of outstanding operations needed to drive high bandwidth. By developing a custom asynchronous user-space HTTP stack and object store library, CPU consumption is reduced to around 3% of Amazon's library. Direct attached storage is used as a cache for blocks on object storage to further increase processing efficiency in the cloud.

Cluster Parity Filesystem

To lower costs while improving availability and reliability for on-premises deployments, Yellowbrick built a stacked, higher-level cluster filesystem called ParityFS. It sits on top of BBFS, implementing the same POSIX-subset interface in an asynchronous, reactive fashion, but provides the the system with resilience against data loss in the event of node failure. In a traditional database, such resilience is provided by "mirroring" or "replicating" multiple copies of data: Typical database deployments will store two or three copies of data across nodes so that, in the event of node failure, another node in the cluster can replace the work of the failed node. However, doing so will lead to that node performing twice as much computation (since 2x more data needs to be processed), leading to substantially slower query processing performance due to the added "skew" in query processing.

Yellowbrick has implemented erasure coding similar to RAID-5 or RAID-6 in a disk-drive configuration but at file level rather than block level. The scheme provides the same level of redundancy as writing three copies of data but without the storage overhead. As files are written in parallel across the cluster, reconstruction data is written. If a node is lost, the files it was storing are virtually reassigned to all the other nodes in the cluster and reconstructed on the fly when read – such that all nodes in the cluster share the processing work of the failed node. The data writing and data reconstruction processes use the massive amount of parallel computation and high network throughput available, such that in the rare event of nodes failing, the overhead added to typical database queries is under 5%. At the time of writing, Yellowbrick production customers that have experienced node failures never even noticed performance degradation.

Query Execution

The Yellowbrick database engine is responsible for taking query plans, performing computation on data – reading and writing as necessary – and returning answers. There are four major software components, all built from scratch by Yellowbrick: The Storage Engine, the Execution Engine, the Workload Manager, and the Query Compiler.

Storage Engine

The Storage Engine stores structured data (two-dimensional tables with rows and columns). It's designed to scale from tables with as few as one row to tables with trillions of rows and thousands of columns, occupying petabytes of storage space. Almost all the data stored in the database is stored in a column-oriented store, and large loads of data are written directly to this store. Recent real-time or streaming data is stored in a row-oriented store instead. When the row-oriented store reaches a certain size, data is automatically moved to the column-oriented store. The storage engine is ACID compliant; transactions semantics are consistent between both the row-oriented store and column-oriented store, and queries automatically look at all the data present in both stores.

Row-oriented Store

The row-oriented store (row store) is a scale-up storage engine. It's optimized for low commit latency for real-time streams, such as those from Kafka or CDC tools. When streaming data (such as INSERT statements in small transactions) arrives in the database, the most important thing for it to do is commit the data as fast as possible and return control to the client so it can continue. The number of real-time, streaming commits per second is thus a function of commit latency, and the fastest path to the lowest possible latency is to avoid networking and commit to disk.

Rows are stored in a log-oriented structure in which multiple files are kept per table per CPU core, and new rows of data are appended in real time. It's stored on mirrored, highly available storage volumes: For cloud instances, on EBS-like volumes; for on-premises instances, on four-way replicated SSDs. When a given table's row store approaches a size where it will have a measurable impact on query performance, it is flushed in the background to the columnoriented store. Users and admins need not concern themselves with this flushing process.

The row store contains a high-bandwidth streaming optimization: When an incoming stream is copying many rows in a transaction without committing, the row store can transparently switch its operating mode to one where it passes the data directly through to the columnstore without writing intermediate data to disk. Upon commit, the data will be persisted to the columnstore instead.

Column-oriented Store

The column-oriented store (columnstore) is where most data in Yellowbrick resides. Columnar databases are nothing new, and the benefits for analytic workloads – improving compression ratios and reducing the amount of scanned data – are well known, so much so that most analytic databases now store their data in columns rather than rows.

Yellowbrick has been creative about how the column-store indexes and finds data efficiently on SSD media and cloud storage. The columnstore runs entirely within the Yellowbrick Direct Data Accelerator®. Static partitioning is not required to achieve acceptable performance: As tabular data is written, rows are distributed across storage nodes in the cluster into multiple partitioned units called shards, which represent around 200MB of compressed data. Data within each shard is laid out in a columnar fashion, so data from individual columns can be read and processed without having to read the whole shard. Rows may be reordered before the shard is written by "clustering" data on as many as four columnar keys. Clustering allows the database to group related rows, such that if, for example, queries often select rows by time and by customer, clustering on both columns will make accessing such data more efficient. Clustering is rarely necessary due to the staggering throughput of Yellowbrick, and won't make much of a difference to large, intensive queries, but it can help you find small amounts of data ("needle-in-a-haystack" queries) even more efficiently by making built-in indexes work better.

As shards are built and written, Yellowbrick builds granular indexes and compresses data. Yellowbrick uses several compression techniques, with a primary focus on the CPU efficiency of decompression rather than using the minimum amount of storage because storage costs are cheaper than compute costs. For each column within the shard, the indexes store common distinct values, a data structure for computing cardinality statistics, and the ranges of values present. These values are stored per shard, and hierarchically within each shard, per 4KBto-32KB disk block, and within that, per small group of rows. This is a far more granular index structure than that used by traditional analytic databases. That's possible because the storage engine has been designed for SSD storage, which excels at large numbers of random disk çoperations (IOPS). When one can execute tens of millions of IOPS while consuming a barely measurable amount of CPU, table scans can be turned into massive numbers of random I/O operations to avoid reading as much data as possible. Doing so allows Yellowbrick to read only the minimal amount of pertinent disk blocks needed for each table scan, but this is done carefully to keep queue depths low enough to minimize L3 cache usage. Achieving massive bandwidth with tiny queue depths requires an extraordinarily efficient IO framework.

Since statistics are stored along with every shard and automatically recombined, there's no need to worry about keeping statistics up to date. Yellowbrick deletes rows in the columnstore by writing deleted row identifiers to new files alongside the shards. It handles row updates by deleting the old rows and inserting new ones into a new shard. A consequence of this design approach is that Yellowbrick is incredibly fast at running large bulk updates and deletes but less efficient at small random ones. A built-in garbage collector periodically sweeps up fragmented shards, removes deleted rows, and recombines them efficiently. In Yellowbrick, it does this incrementally in very small units of storage, so that – unlike in older databases where performance suffers greatly during vacuuming – there is no measurable impact. These processes do not need to be initiated by a DBA.

The architectural approach means that over time, for update-oriented and delete-oriented workloads, shards will retain garbage (old rows of deleted data) and storage efficiency will drop. However, the benefits of writing and recombining immutable shards of data enable us to implement functionality like data snapshots and time travel relatively easily. The former is already productized for the backup and restore functions, and the latter is a committed roadmap item.

Locking and Transaction Management

Yellowbrick, being an enterprise-class database, implements full ACID transactions. The isolation level provided universally

is READ COMMITTED. Locking is performed at the table level. The Yellowbrick transaction log comes from PostgreSQL and is solid, with decades of production use. Locking is built using a multi-version concurrency control (MVCC) approach. Regardless of how much updating, deleting, or loading is taking place, readers still see the data as of the transaction they are in. Readers never block each other. Generally, writers block other writers and will queue behind each other. However, there are isolated cases where more than one writer is allowed into a table:

- Multiple "bulk loads" can run concurrently into one table.
- Bulk loads can run concurrently with other update and delete operations.
- Data flushes from the row store into the columnstore can run concurrently with bulk loads.

Hybrid Execution Engine

Just as the database stores data in both rows and columns, Yellowbrick also executes queries in a row-oriented or column-oriented fashion. The execution engine, otherwise known as the "EE", is modeled after packet-processing frameworks, just like networks. SQL operators are like nodes in the network, and packets flow over the links between the nodes.

Query Topology and Flow Control

A query planner turns a SQL query such as:

```
SELECT a,b FROM foo INNER JOIN bar on foo.p=bar.f
```

into a hierarchy of SQL operators. In the case of this simple query, the query plan contains two table scan nodes and a join node. Data flows from the bottom of the tree up to the top of the tree, where it's returned to the user.

In the EE, queries are represented by graphs rather than trees. The nodes in the query graph are the operators – such as table scan, join, or sort – and the edges connecting the operators are links. Graphs allow us to plan and execute complex query topologies, such as a table scan that feeds multiple consumers of data at the same time. The job of the EE is to execute the query graph optimally. Data flows from the leaves of the graph toward the root of the graph. As data is handed between nodes in the graph, it's placed into packets that may contain row-oriented or column-oriented data. Packets are sized to make optimal use of L1 and L2 cache memory, so they can be kept core local as they move between nodes. Flow control is required, just like in networks, to ensure that memory usage and queue depths of packets are bounded to stay cache-resident. This is because some nodes may produce far more packets than they consume, such as an outer or cross join, whereas others may produce far fewer, such as an inner join with few matching rows.

In contrast to traditional databases that use "iterators" to pull data from the top of a tree (the "volcano model"), Yellowbrick uses a more sophisticated approach that lets us tightly control resources, where grants are handed toward the leaves and data flows in the reverse direction.

The Distribution operator, which moves data packets across the physical network between MPP nodes, also uses the same flow control and backpressure approaches – in essence, extending the EE graph to be global across MPP nodes.

The network buffers are the packets themselves and they can be transmitted and received in place with no data copying when supported by the underlying YRPC transport.

The entire EE framework is fully multi-core and NUMA-aware. Wherever possible, 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 packets and operators to cores and NUMA nodes also extends across the MPP network.

Row-oriented and Column-oriented Execution

It's optimal to process data in different ways depending on the type of SQL operator node in question. The EE supports row-oriented and column-oriented packets. For example, the Distribution operator, which moves data packets across the physical network between nodes to implement MPP execution, wants to operate in a row-oriented fashion because rows of data will be transmitted to different MPP nodes depending on the hash of a column in the row. Likewise, the Join operator combines rows from two different tables and concatenates them.

On the other hand, the table scan node prefers to operate on columnar data straight from the Storage Engine, where it can take advantage of vectorized execution. Vectorization allows us to apply efficient SIMD (AVX) instructions to build incredibly efficient predicate filters, expression calculators, or bloom filters that can operate on multiple values in one CPU instruction, but it requires that data is laid out in a manner that is amenable to the instructions.

Partitioning

Like many OLTP and OLAP databases, Yellowbrick supports a SQL syntax for partitioning tables. Admins can define partitioning schemes on a per-table basis. Currently, hash partitioning and range partitioning are the available schemes.

Most MPP databases implement partition pruning at the planner level to find data more rapidly by sub-setting the data that needs to be scanned. In Yellowbrick there's no need for partition pruning because shard indexing is so efficient. Instead, all knowledge of partitions is pushed down to the executor itself. DBAs configure Yellowbrick to use partitioning solely to reduce memory usage for massive joins or aggregates. Consider two examples:

1. A hash join of two massive tables with perhaps hundreds of billions of rows (a fact-to-fact join). The traditional
2. approach would be to build a hash table on the smaller side of the relation and scan the larger side, looking up each row and emitting the joined result. The hash table is still huge, occupying a massive amount of memory and likely forcing the query to spill.
3. A billing query for complex call data record (CDR) time-series data, with more than 1 billion subscribers, summing call costs by phone number and call. For a large telecom operator with hundreds of millions of subscribers making large numbers of calls per day, the hash table for the GROUP BY operator would surely be gigantic and spill to disk.

In the case of example 1, if both hash tables are partitioned identically, the join can be executed a partition at a time because it's known that the data in the partitions is non-overlapping. If 1,000 partitions were chosen, 1,000 smaller joins

would be performed rather than one massive join, using 1/1000 of the necessary memory. The query will execute faster and will not spill. Similarly, for example 2, rather than running one giant aggregate, if it is known that the CDRs have been partitioned by hour, the query can run one aggregate per hour and incrementally emit the result.

Yellowbrick accomplishes this by adding partition iterator nodes to the query plan. Partition iterators repeatedly reset and re-run all downstream operators, once for each possible partition value, incrementally emitting results. This is a far more sophisticated approach than naive planner partition pruning and yields substantial benefits in runtime efficiency by reducing memory utilization and eliminating spilling for many complex queries.

Storage Predicate Pushdown

The Yellowbrick Storage Engine maintains various granular indexing structures to efficiently find data that's necessary and avoid data that isn't. For example, if you run a simple query:

```
SELECT * FROM person WHERE age < 24;
```

The executor will pass the predicate age < 24 to the storage engine to enable it to return only rows matching the criteria.

Yellowbrick does both static predicate pushdown – identifying predicates derived during query planning such as described above – as well as dynamic predicate pushdown, which adds additional predicates to queries at runtime. Static predicates are identified at plan time, but the actual values are injected at runtime to enable parameterized queries to make use of pushdown functionality.

Runtime predicates are generated by joins as well as SQL constructs that perform similar operations to joins, such as large IN lists or sequences of OR criteria, which are internally rewritten to semi-joins. The runtime predicates typically take the form of bloom filters, pushed down after generating the build sides of hash tables. Yellowbrick also creates BETWEEN predicates from the minimum and maximum values present in the build sides of joins, which helps in surprisingly common cases where there is correlation between the two tables.

Query Compilation

All queries in Yellowbrick are aggressively compiled to CPU instructions to run as fast as possible, in their entirety. Yellowbrick contains no interpreter and no just-in-time compiler for queries. Memory management in queries is explicit, with no garbage collection. Yellowbrick contains a SQL compiler built from scratch called Lime. Lime's job is to consume the output of the query planner and generate code to execute the query. It understands the Execution Engine and the reactive programming model. Lime's processing consists of multiple phases:

- Produce an abstract syntax tree (AST) for the incoming query plan, converting query plan nodes to execution engine operators.
- Perform a type optimization phase.
- Apply several optimization passes on the AST: rearranging data to be contiguous in memory, static evaluation, reordering of memory accesses, fast-path deduction, and so on.

- Emit code (including inline assembler) corresponding to the AST.
- Compile the code using the optimizing LLVM compiler infrastructure.

LLVM Compilation

Lime uses a modified version of the LLVM compiler infrastructure to generate machine code. LLVM itself can generate highly optimized code. The EE framework and SQL operator templates are built in C++. They use inline-able injector functions: It's not just the "core loops" of the operators that are compiled; the code implementing the entire operator is ultimately expanded to one set of code that can be aggressively optimized by LLVM.

This is a relatively expensive process, especially in a single-threaded compiler such as LLVM. To provide interactive queries quickly, Lime will split each query into multiple exclusive segments – segments for which in-lining code from other segments would add no value – that can be both generated and then compiled in parallel across multiple CPU cores. To further lower latency, we have modified LLVM to allow it to remain memory-resident in a state with its own ASTs preloaded. If compilation becomes a bottleneck, the compile service itself is elastic and horizontally scalable, and more pods will be spun up.

This parallelization allows simple queries to be compiled and executed in milliseconds, while very large, complex queries can still compile within a couple of seconds, fast enough for interactive analytic queries.

Pattern Compiler (Regular Expressions and Friends)

Regular expressions and LIKE operations have historically been slow when run against large data sets. To improve execution speed, Yellowbrick implements a special compilation framework called the pattern compiler. The pattern compiler currently supports the following input patterns:

- SQL LIKE
- SQL SIMILAR TO
- POSIX-compatible regular expressions
- Date/time parsing

The pattern compiler generates highly optimized, deterministic finite state machines for each unique pattern. The set of state transitions is optimized and compiled into optimal machine code by the LLVM compiler infrastructure and then loaded on the fly into running SQL queries. All POSIX regular expression functionality is supported, including all character classes, operators, and capture groups. The pattern compiler supports backtracking and analyzes subexpressions to determine whether a faster deterministic, or slower nondeterministic, finite automaton, is necessary on a per-operator basis. Storage predicate pushdown is performed for the starting characters of all patterns.

Yellowbrick has one of the fastest database regular expression implementations ever created, if not the fastest. At the time of this writing, the Yellowbrick database contains no support for interpreting or doing JIT compilation of patterns so it's not possible to store a regular expression in a table column and use it in a query; however, you may supply patterns as runtime parameters in queries.

Code Caching

Although code compilation is fast, latency is still minimized by saving milliseconds of compile time for short, tactical queries and seconds of compile time for large, complex ones. To do that, the query compiler contains multiple layers of caching. Efficient and reliable caching requires care and attention: A given query has a huge quantity of dependencies, from execution engine templates to the version of the C++ runtime, versions of libraries, the query plan in use, the statistics of the query, and so on. Yellowbrick factors all these dependencies to try to reuse as much previously compiled object code as possible.

PostgreSQL Compatibility and Query Planning

We use open-source technology for external access to the database. PostgreSQL is the logical choice: Its SQL support is very close to ANSI standard, with a track record of adoption in other data warehousing platforms (Amazon Redshift, IBM Netezza, Vertica, and so forth). Furthermore, PostgreSQL has become perhaps the coolest and fastest-growing relational database, with a thriving ecosystem of developers and users contributing to its success.

Compatibility

The front end of the Yellowbrick database derives from PostgreSQL 9.5.x. Yellowbrick periodically merges fixes and enhancements from newer PostgreSQL versions where it makes sense. Yellowbrick is not a PostgreSQL fork; if you write an analytic query against Yellowbrick, more than 99% of the machine instructions will be running against the new Yellowbrick code. The core parts of PostgreSQL – many parts of the query optimizer, the storage engine, and the execution engine – have been entirely replaced in Yellowbrick.

Access to Yellowbrick is typically accomplished via standard PostgreSQL ODBC/JDBC/ADO drivers. This is a deliberate choice because it allows the database to interact with numerous ecosystem tools. That said, PostgreSQL's wire protocols are not particularly efficient, so Yellowbrick is actively developing custom higher performing *DBC drivers while continuing to maintain Postgres compatibility. PostgreSQL's metadata catalogs are also supported for interoperability with standard tools and will be familiar to admins and developers alike.

SQL Dialect

At the time of this writing, Yellowbrick supports the following SQL data types: Booleans; integer types; decimal types; floating point types; UUID, CHAR, and VARCHAR; date and time types; and some new data types for IP addresses and MAC addresses. INTERVAL is supported as an intermediate data type and cannot be stored. TEXT is present for PostgreSQL compatibility and internally aliases to VARCHAR(64000).

PostgreSQL's SQL dialect has been extended with functions for Oracle, Microsoft, and Teradata compatibility, and a new, SQL-based user-defined function (UDF) grammar. Stored procedures written in PL/pgSQL are supported. Partitioning is supported. Advanced PostgreSQL functionality such as XML, JSON, geospatial SQL, and other programming languages for server-side programming (Python, Perl, TCL, and so forth) are roadmap items.

Numerous utility commands have been added to the SQL grammar for tasks such as managing the workload management subsystem, controlling elastic clustering, and loading and unloading data from external sources. You can find a full description of Yellowbrick SQL support in the product documentation.

Query Planning

Yellowbrick has replaced, extended, and altered many parts of the PostgreSQL query planner. Query planning is a large and complex subject with entire papers devoted to individual optimizations. Here's a partial list of examples in Yellowbrick:

- MPP awareness, with cost-based estimation of network data exchange.
- MPP plans for all standard primitives (joins, aggregates, sorts, distinct counts, and so on) and data-distribution nodes for hash-distributed, replicated, and randomly distributed data sources.
- Join algorithm replacement, with a new estimator and costing model that uses metadata, histograms, implied equalities, and uniqueness inference.
- Scan selectivity algorithm replacement.
- Support for various forms of correlated subqueries.
- Support for using and combining incremental statistics using big data algorithms.
- Query auto-parameterization to support plan re-use.
- Support for partitioning and adding partition iterators to query plans.
- New data type hierarchy to match other enterprise databases and remove the need for excessive explicit type casting in expressions.
- Late-bound views, wherein views are evaluated when accessed, so database objects referenced by views can be dropped and recreated.
- Support for expression aliases.
- Normalizing in-lists, semi-joins, and OR-lists.
- Support for planning graphs rather than just trees.
- Reimplementation of estimation for aggregates.
- Filter push-down improvements for table scans.
- New EXPLAIN interface.
- Constant folding.
- Static elimination of relational operators and expressions.

Changes to the security model to eliminate "super users" have also affected the query planner to some degree; see the Security, Systems Management, and Monitoring section for details.

Query Processing and Workload Management

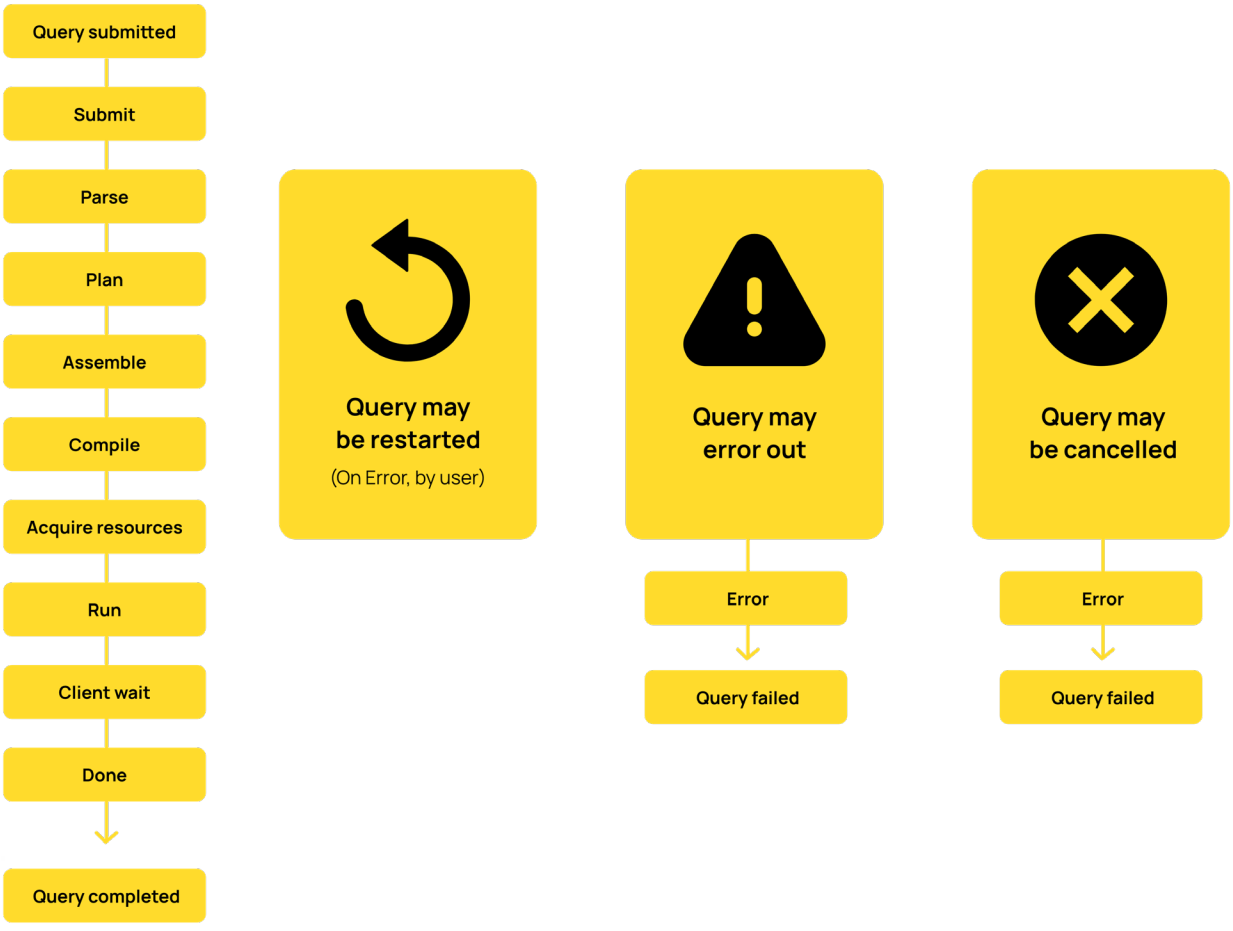
When a user submits a SQL command that involves query processing – henceforth referred to as a "query" – to Yellowbrick via ODBC/JDBC/ADO.NET, the query is passed through several subsystems in its journey through the Yellowbrick database software stack. At the bottom of the stack, the query ends up in the Storage Engine and Execution Engine, at which point it generates results. The results are then passed back up the stack and returned to the user.

During this journey, what starts as a simple piece of SQL text is enriched with more metadata and information. The workload management subsystem is present throughout the lifetime of the query, allowing it to be measured, monitored, routed, prioritized, and adjusted as it travels down and back up the software stack. The adjustments relate to both the query itself, as well as to limits on resource consumption. The Workload Manager allows DBAs and users (with prerequisite permission) to see what's going on as well as to change the nature of the routing to ensure business goals are met. For example:

Certain business-critical reports may be a higher priority than anything else going on in the system.
"Bad actor" rogue queries, perhaps written by users with poor working knowledge of SQL, should not affect other users.
ETL tasks shouldn't consume more resources than necessary.

Query Processing Flow

Each query goes through the state transitions shown in Figure 2. The meanings of these stages are generally self-explanatory. When a query is submitted, little is known about it: its text, who submitted it, which machine and client it came from, and when. After parsing, a query is planned by the query planner, at which time a lot more information is known: which tables it looks at, its cost, and estimates of resource consumption. It's at this point that any table-level locks necessary to execute the query are acquired. Query fragments are assembled and then compiled to native code by the LLVM compiler infrastructure, at which point the query may wait to acquire the necessary resources (memory, disk, processing power) to execute. The query then runs, at which point it sends results back to the client and is then marked as done. Queries can be canceled or moved into an error condition at any point in the flow.



Monitoring and Introspection

The Workload Manager is the single source of truth for everything executing in the database. It records, for every query in the system, when that query entered a given processing state and how much time it spent there. Through system tables, users can view this information for all queries currently executing in the database as well as previously completed ones.

As one would expect for an enterprise-class database, runtime statistics are measured for every query. Statistics include how much CPU was consumed, how much IO was driven, how many rows were processed, and the sizes of data structures such as hash tables in joins. These statistics are logged on a per-query as well as a per-query-plan-node basis and can be consumed raw in the database or graphically in a web browser via Yellowbrick Manager.

Internal Details

Yellowbrick's Workload Manager can proactively allocate and manage the following system resources:

- Persistent storage space – through disk quotas.
- Spill space.
- Memory.
- CPU – through priorities.

Resource Pools

Each Virtual Compute Cluster's available resources are divided into pools. Queries are routed to a given cluster's pool by rules executing within the Workload Manager (see Rules below). There may be, for example, a small pool for DBAs to make sure they are always able to acquire sufficient resources to kill queries, a pool for short-running, tactical queries, and a pool for long-running ones.

A pool has associated with it a certain level of concurrency it can support, which may be fixed or flexible. Fixed concurrency is useful when it is known that only a certain amount of concurrency is possible or desired, whereas flexible concurrency is a good idea for random queries by data scientists. Queries slow down as they spill more, and it is desirable to have queries run faster when there are fewer users on the system, and slower when there are more users on the system.

Rules

At various points in the execution of a query, Workload Manager rules can introspect what's going on, and perform various actions: cancel the query, route the query to a different pool, prioritize it, or alter its resource consumption. A query can also be throttled according to a named semaphore – for example, to limit every business user to submitting at most two concurrent queries or ensure a misconfigured ETL tool doesn't overload the system with concurrent singlerow retrievals.

Workload Manager rules are written in JavaScript, providing incredible flexibility. Properties about the system and each executing query are available via standard JavaScript properties, and actions such as pool assignments, throttling, logging, resource assignments, and recording errors can be accomplished through JavaScript methods. To reduce the runtime overhead of evaluating rules, all the JavaScript rules are compiled into native code for rapid execution.

Some DBAs are not comfortable writing JavaScript, so within the browser-based Yellowbrick Manager is an intuitive point-and-click IF/ELSE rule builder for the Workload Manager. The JavaScript generated by the rule builder can be seen alongside and tweaked if desired. For users who prefer not to use a GUI, a full SQL utility grammar is present for creating, modifying, and controlling workload management rules.

Control Points

Many different types of rules operate at different control points in the system. Typical deployments don't need to implement numerous rules or sophisticated behaviors, but some customers with massive, operational warehouses that support multiple lines of business tend to make maximum use of the flexibility:

- Submit rules are run before queries enter the parser. They allow queries to be rejected by rules that allow matching by SQL text or source IP address.
- Assemble rules are evaluated when the query is building the artifacts required for query compilation before it is submitted for compilation. A common task at this stage is to set the initial priority of the query which helps govern progress and resource usage in later phases.
- Compile rules run during the pre-execution phase for a query, prior to when a resource pool being selected. Rules written for this stage of execution have more information from the resource planner and are commonly created at this phase for pre-execution concerns. Various actions can be taken at this stage, including imposing limits on execution time and memory, setting query priority, or selecting a resource pool.
- Restart rules are evaluated when a query encounters an error condition that allows it to be restarted automatically – for example, to re-run it with more resources or log the event.
- Runtime rules execute periodically during query execution and can introspect the state of the query along with its run-time statistics: how long it's been executing, what resources it's consumed, or the state of its query plan. At runtime, queries that haven't returned data to the user can be transparently moved between pools – perhaps with differing priority or resource limits – or rejected. SNMP traps and alerts can be generated for external systemsmanagement tools. This is how you can be sure that misestimated, long-running queries don't get in the way of tactical ones, or that "bad" queries are placed in a penalty box such that they don't affect other users.
- Completion rules are evaluated once when execution steps are completed or stopped, due to the query running to completion, being canceled, or erroring out.

Logging capabilities are available to trace rule execution. Rules have priorities and are evaluated in priority order.

Profiles

A container of rules is called a profile; profiles may be imported and exported to/from JSON to move them between instances. Each elastic compute cluster has an active profile that can be changed at any time. If, for example, an insurance

company needs to prioritize workloads differently for closing the books at the end of the month, it can switch profiles to enable this only for the days necessary.

Availability and Business Continuity

Yellowbrick has been designed for high availability and a degree of fault tolerance, and with support for backing up and restoring data, replicating data, and storing data on cloud object stores. No third-party tools, services, or "enterprise editions" are needed.

Yellowbrick is designed for data warehousing and analytics rather than online transaction processing, so "five nines" of availability – 99.999% uptime – is not a requirement. Five nines allow only about five minutes of downtime per year and necessitate no maintenance windows, fully online upgrades, and the like. Yellowbrick's architecture is centered around achieving "threeand-a-half nines" per instance – 99.95% availability – allowing about 4.5 hours of downtime per year. This allows time for scheduled downtime for short quarterly maintenance windows to support software upgrades, along with a small amount of unscheduled downtime due to infrastructure failures or occasional software bugs. Yellowbrick doesn't claim to have mainframeclass or Oracle-class stability and availability, but its track record is excellent: Yellowbrick routinely backs production, online, ad hoc 24/7/365 business-critical financial applications for many Fortune 500 customers with minimal unscheduled downtime.

Yellowbrick's storage and compute can scale elastically; resizing clusters, creating new ones, or adding storage capacity (for on-premises instances only) are all online activities that do not require maintenance windows.

High Availability

Yellowbrick is clustered software, running on multiple server nodes. The software is redundant to component failure, drive failure, and entire node failure with minimum user disruption. Running in the cloud, failed nodes are replaced by new ones as they become available; on-premises, failed nodes are left out of the cluster until a replacement is installed. For more details about onpremises hardware instances, see the "Andromeda Optimized Instances" whitepaper.

Protection for Data Stored in Cloud Object Stores

For instances running in public clouds, as well as on-premises instances accessing cloud storage, all column store shard files are persisted to cloud object stores. In Amazon, this means S3; in Azure, ADLS; and in GCP, GCS. A fraction of local direct attached SSD storage is used as a block-level (not file level) cache for data persisted in object stores. The block cache is scanresistant such that large table scans where data is only used once will not evict hot data from the cache. Cache misses result in the given block/s being requested from the cloud object store directly. Due to the massive parallelism needed to achieve good read performance from object stores, table scans will do a substantial read-ahead. For writing table data, shard files are written through to the cloud object store, and the transaction is only committed once the object store has acknowledged the write.

Protection for Data Stored in On-premises Instances

Yellowbrick's cluster filesystem ParityFS protects against data loss due to node failure. It implements n+2 erasure coding, such that two nodes in every parity group can be lost in their entirety due to node failure, or partially due to SSD failure. Data is reconstructed on the fly, and when drives or nodes are replaced, the original data is rebuilt. The difference in query execution performance and throughput is unnoticeable; failure of a node or SSD results in a cluster reconfiguration event which causes momentary service degradation.

Backup and Restore

The Yellowbrick Storage Engine can take transactionally consistent snapshots of all the data in a database at any time. These snapshots are low overhead, leverage the transactional nature of the database, and are completed in a fraction of a second. The snapshot metadata itself occupies very little storage; however, active snapshots impose constraints on garbage collection such that the system will use more storage space until the snapshots are dropped.

The snapshots are per-database and include all tables, including system catalog tables: Those that store the database schema objects such as tables, columns, and constraints; as well as roles, workload management rules, and other system configurations. In the roadmap, the snapshot functionality will be exposed to support "time travel" queries for general SQL use.

SQL-native Backup and Restore

Backup and restore operations are implemented as SQL operations to leverage the high throughput of the Yellowbrick database engine. By performing delta queries against transaction snapshots, the system can select only the rows that have been added, updated, or deleted between two transactions for backup. Changed rows are then compressed in parallel on all nodes for increased performance; backups are not simply rigid copies of files in the filesystem.

Restore is also implemented as a SQL operation and is analogous to bulk data loading. All nodes in the cluster receive backup data. It is decompressed and decoded in parallel on all workers and new data/changed data is written to the Yellowbrick Storage Engine.

Table Delete Horizon

The history of different types of database backups – full, incremental, or cumulative – together forms the backup chain for that database. Each chain is a logical grouping of snapshots, each representing a discrete point in time. As a result, each backup chain has an implicit "delete horizon," a transactionally consistent point in time for the database after which deleted space cannot be fully reclaimed. Deletes made after the horizon point in time leave behind "tombstone" markers in the storage engine that occupy tens of bytes per row. Backups move the delete horizon forward in a way such that deltas of previous backups can still be taken.

Types of Backup and Restore

Backups work one database at a time. Each backup requires the creation of a new snapshot. Yellowbrick supports three types of backups:

- Full backups are complete backups of an entire database. A full backup is typically done once, to initiate a new backup chain and rarely thereafter. Full backups tend to be very large and can be thought of as a set of all changes between the first transaction and the backup transaction snapshot.
- Cumulative backups capture all the changes since the last cumulative or full backup snapshot, whichever is more recent, and advance the table delete horizon.
- Incremental backups capture all the changes since the last incremental, cumulative, or full backups, whichever is more recent, but do not advance the table delete horizon.

Yellowbrick also supports incremental restores so that deltas captured by an incremental or cumulative backup can be applied without having to restore the entire database. For the data and schema, this is straightforward; for other parts of the system catalog, various merge strategies are used. More advanced backup strategies are possible but have not been implemented.

Readable Replicas

A hot standby database is a database that is in read-only mode while receiving continuously applied incremental restores. Yellowbrick allows hot standby databases to be queried and used, including the creation and deletion of temporary tables as needed by reporting and BI tools. In addition, a database can be placed in a purely read-only mode to "freeze" an environment.

Asynchronous Replication for Disaster Recovery (DR)

Yellowbrick contains full support for unidirectional asynchronous replication, to be used for establishing a DR site. Replication can be between on-premises and cloud instances, and across cloud vendors, as desired. No third-party utilities are required for replication. Both DDL and data are replicated. Replication takes place over TLS-secured TCP sockets and is interruptible: In the event of socket failure, the replication process will pick up roughly where it left off and continue from there. The target databases for replication must be hot standby databases, enabling them to be actively used for queries while receiving writes.

The replication process does place some additional query load on both databases. Replication is transactionally consistent, with data written to the target in one transaction to guarantee consistency for users. Where possible, an initial replication target should be seeded using backup-and-restore functionality. Replication is configured, managed, and monitored through full SQL utility grammar and system tables. Currently, replication is supported from one source database to one target database. Multi-target support is not currently productized.

Failover and Fail-back

In Yellowbrick, failover is not automated because it is considered a rare event. Individual instances are highly available and protected, so failover is only necessary in the event of a mass loss of connectivity or a true natural disaster. The database will ensure that no split-brain scenario has occurred whenever replication takes place by checking transaction sequences and database configuration between the source and target. If inconsistency is detected, manual intervention is required. Fail-back after failover is fully supported but note that bi-directional replication, where both databases are accepting changes, is not supported.

High-throughput, Parallel Data Movement

Data analytics platforms need to be able to ingest and unload large amounts of data rapidly. Sometimes data ingest and extraction needs to be done in batch mode with as much throughput as possible; whereas other times, the data movement needs to be done in a streaming fashion for real-time queries. Sometimes the data movement is best initiated through SQL. Sometimes it's taken care of by modern pipelines built with Kafka and/or Spark. And in traditional deployments, it's initiated as part of a complex set of scripts involving ETL tools. Yellowbrick caters to all these scenarios.

Bulk Data Load and Unload

Yellowbrick contains an efficient binary protocol for parallel bulk loading and unloading. Data is streamed to or from the database in parallel, across multiple network sockets (typically one socket per node in the cluster). The bulk protocol is row-oriented and supports compression.

These protocols are designed for batch operation. Large unloads complete in one shot, and large loads will commit a few massive transactions periodically. Format transformations such as parsing and formatting are done by the sender (for loads) and receiver (for unloads) to reduce load on the database. Bulk operations pump data directly from/to the Yellowbrick Storage Engine.

These operations typically are bound by the speed of the network. The ways of initiating bulk data movement with Yellowbrick are:

- Through SQL: Using the built-in utility statements such as CREATE EXTERNAL STORAGE, CREATE EXTERNAL FORMAT, CREATE EXTERNAL LOCATION and LOAD TABLE.
- Through pre-existing integrations with products such as Kafka, Spark, Informatica, Oracle Golden Gate, etc.
- Through traditional cross-platform client tools that run on everything from Windows to AIX: ybload and ybunload.
- Through Java integrations with a Yellowbrick-provided data loading library.

Progress of loads and unloads, regardless of whether they are initiated using client tools or SQL, can be monitored through system views. A variety of file formats are supported including Apache Parquet as well as traditional delimited data and BCP files. Yellowbrick Manager also enables browsing and loading of data through a web browser.

Streaming Data Movement

Because bulk data is committed in large chunks, and socket negotiation across clusters is required to initiate the process, it's an inefficient way to load and unload small numbers of rows. For small unloads, the ODBC/JDBC/ADO.NET (henceforth "*DBC") drivers will suffice, and PostgreSQL's built-in \copy command is a good shortcut.
For loading small numbers of rows in a streaming fashion, you can use *DBC INSERT statements, or alternatively, \copy. With a few parallel clients, when loading data directly into the roworiented store, rates of several million rows/sec are achievable. Unlike bulk loads, which are committed in huge batches of hundreds of millions of rows, row-store transactions can be committed frequently – even once per row, if desired.

Unlike other data platforms that struggle to do streaming ingest efficiently, the hybrid nature of the Yellowbrick Storage Engine allows it to easily ingest large numbers of small transactions and query the most recent data, along with historical data, in a transactionally consistent fashion. This is ideal for integrations with CDC tools (Oracle Golden Gate, HVR) or an enterprise message bus (Kafka). There's no need to worry about micro-batching or writing custom code to combine reporting from transactional and analytic databases.

Concurrent Loading and Querying

Yellowbrick can efficiently query tables as data is being written to them. Large bulk loads or data merges running in parallel will not alter query performance in any unexpected ways. Consequently, it's possible to have databases with a very high level of churn that can handle large numbers of concurrent, ad hoc queries at the same time. Some Yellowbrick customers have databases hundreds of terabytes in size where each day 30% of the data continuously changes.

Security, Systems Management, and Observability

Enterprise-class databases must be secure, and easy to manage, monitor, and configure. Yellowbrick has a variety of mechanisms that cater to the needs of enterprise customers running in private and public clouds, in both HIPAA-compliant and regular deployments.

Security

Yellowbrick is built assuming everything is "private by default." Default access controls to areas of the product are limited. Private S3 buckets are enabled by default, with no public access to data. No built-in guest user accounts are present, and strict access must be granted to all data and management functionality. Because the Yellowbrick runs in a customer's own cloud account, Yellowbrick by default has no access to customer data, customer queries, or logs. Companies have full control of who has access and what they can access.

Authentication

Yellowbrick's authentication is based on OpenID Connect (OIDC) and supports authentication with all OIDC-compliant identity providers such as Azure Active Directory (aka Office365), Okta, and Ping. OIDC authentication allows companies

to continue to maintain identities and roles in their existing identity providers making the management of users easier to maintain. On-premises Yellowbrick instances can authenticate with LDAP and Active Directory, also synchronization of roles between external identity providers and the database is supported. For business-critical activities, the Yellowbrick database supports local user authentication to allow operations to run in case connectivity to external authentication is unavailable.

Manageability Without "Super Users"

PostgreSQL relies on having a "super user" for administration and regular users for everything else. The super user can administer anything whatsoever, much like the "root" user on a Unix system. Although the front end of the database is based on PostgreSQL, privileges afforded to the super user have been deliberately split into dozens of different grants to allow users to manage subsections of the database in a far more fine-grained manner – for example, the ability of a role to manage other roles, view SQL query text of other users, initiate backups, control LDAP integrations, and even the ability to grant privileges themselves all can be granted or revoked individually.

Role-based Access Control

Role-based access control allows mapping roles to specific database access controls. Roles are granted and revoked using already existing identity provider management tools, such as adding a role to a user in Active Directory. Access to all database schema objects is fully role-based. Roles can be used to grant access down to column granularity. PostgreSQL administrators will be familiar with this concept.

Encryption of Data at Rest

Encryption of data on cloud object stores is managed by the object store itself and ephemeral cloud storage is also encrypted and crypto-erased. Yellowbrick on-premises instances store all data fully encrypted with AES-256 using keys stored in HashiCorp Vault.

Column-level Encryption and Functions

Yellowbrick provides a variety of SQL functions for data encryption, decryption, and hashing using several algorithms. Individual VARCHAR columns within tables can be designated as encrypted so that Yellowbrick will encrypt the data for the column as it's inserted. When a user with access to the corresponding encryption key executes a query, they will see the decrypted data; however, users without access to the encryption key will see only the encrypted, scrambled data. Keys are stored in and referenced from the built-in HashiCorp Vault.

Yellowbrick has a production-quality integration with partner Protegrity which provides sophisticated, policy-based data protection and masking functionality that goes beyond Yellowbrick column-level encryption.

TLS Support

All communication over the wire is encrypted. TLS 1.2 is required for all traffic in and out of Yellowbrick. This is true for *DBC access and web access and all external connectivity, loading, unloading, backup, and so on. TLS mutual authentication is used for authentication of crossdatabase replication sessions.

Observability

Instrumentation and statistics gathering in the database are visible through system views. A number of additional integrations are available to allow the database to be observed, monitored, and alerted upon in customer environments.

Instance Observability

Yellowbrick Manager includes an observability stack consisting of Prometheus, Loki, and Grafana. These cloud-native technologies are used to aggregate logs from the database, delivered via FluentBit, and raise alerts in the event of issues being found. Prometheus alertmanager is used to deliver alerts to such typical receivers as Slack, Opsgenie, PagerDuty, etc. On-premises installations of Yellowbrick are also able to deliver alerts via SNMP and logs through remote syslog.

User-configured alerts can be raised by the Workload Manager and are raised automatically when user disk quotas are exceeded. Other alerts pertain to unexpected errors in the database, such as software crashes or replication falling behind.

Remote “Phone Home” Support

Yellowbrick is designed with a SaaS user experience in mind, regardless of whether the database is running in a private data center or the public cloud. Yellowbrick optionally maintains remote, unidirectional connectivity to Yellowbrick’s internal SaaS monitoring platform. Any crash minidumps, key telemetry, and key logs are sent back over this phone-home connection to enable Yellowbrick’s Customer Success team to monitor customer systems 24/7. In the event of any issues, Yellowbrick will know about them first. Different levels of data scrubbing guard personal identifiable information (PII) and other confidential data. No customer data is ever shared. Customers with strict security requirements can disable Phone Home completely.

Summary

This whitepaper describes how Yellowbrick rebuilt the cloud database software stack by leveraging Kubernetes and modern computer architecture to implement a modern, fully elastic SQL analytic database with separate storage and compute. By revisiting key assumptions in database architecture, Yellowbrick delivers the most efficient data platform in the industry with flexible deployment in customers’ own cloud accounts or on-premises.