Yellowbrick

Yellowbrick Data Warehouse Architecture

April 2021

Table of contents

Introduction 2
Evolution and impact of computer architecture 4
Introducing the Yellowbrick Kernel 7
Query execution 11
PostgreSQL compatibility and query planning 21
Query processing and workload management 24
Business continuity 28
High-throughput, parallel data movement 33
Security, systems management, and monitoring 36
Summary 39

Introduction

Yellowbrick Data Warehouse is an advanced, massively parallel processing (MPP) SQL database designed for the most demanding batch, real-time, ad hoc, and mixed workloads. It can run complex queries at up to petabyte scale across numerous nodes, with guaranteed sub-second response times.

Yellowbrick was conceived with the goal of optimizing price/performance. New SQL analytics use cases are emerging all the time, and more concurrent users are consuming more ad hoc analytics. That requires more performance per dollar spent, and Yellowbrick architecture leapfrogs the industry in this respect. It's not uncommon for customers to see their workloads run tens or hundreds of times faster at a fraction of the cost compared to cloud-only or legacy data warehouses.

At Yellowbrick, we aren't interested in incremental improvements in efficiency, however. Incremental is boring! Rather, our goal is to make step-function improvements in economics, and when it comes to data processing, these improvements come from modern hardware technologies that are more efficient than traditional systems. For example:

- Machine learning: Accelerated by dedicated processors from Google, Nvidia, Amazon, and others using new memory technology called High-Bandwidth Memory (HBM)
- Storage: Accelerated by the move from hard drives to SSDs, and new fabrics
- Networking: Accelerated by chips such as Amazon Elastic Network Adapter (ENA) and OS bypass
- Web search: Accelerated by FPGA processors
- Bitcoin mining: Accelerated by Bitcoin ASICs

At Yellowbrick, we continuously implement hardware advances similar to those above (e.g., NVMe and flash memory), as well as advances in software (most recently Kubernetes), in our adaptive "cut-through" architecture. We combine these advances with smart thinking about storage formats and indexing, and add on top a modern, standards-based database interface that's familiar to users (PostgreSQL) for ecosystem compatibility.

The result is a modern, quickly provisioned, and easy-to-use data warehouse that knocks the socks off rivals in price/performance economics and can be deployed across distributed clouds (private, public, and edge networks), including within customers' own VPCs—with all instances, databases, and users managed via a simple, unified control plane (Yellowbrick Manager).

This white paper describes the architecture of Yellowbrick Data Warehouse and how new thinking in the context of modern instances can bring about truly creative solutions to old problems such as storage management, execution performance, concurrency, speed, and cost. (See Figure 1 on next page for high-level diagram.) We are proud of how far we've advanced the state of the art and are excited to share our journey!

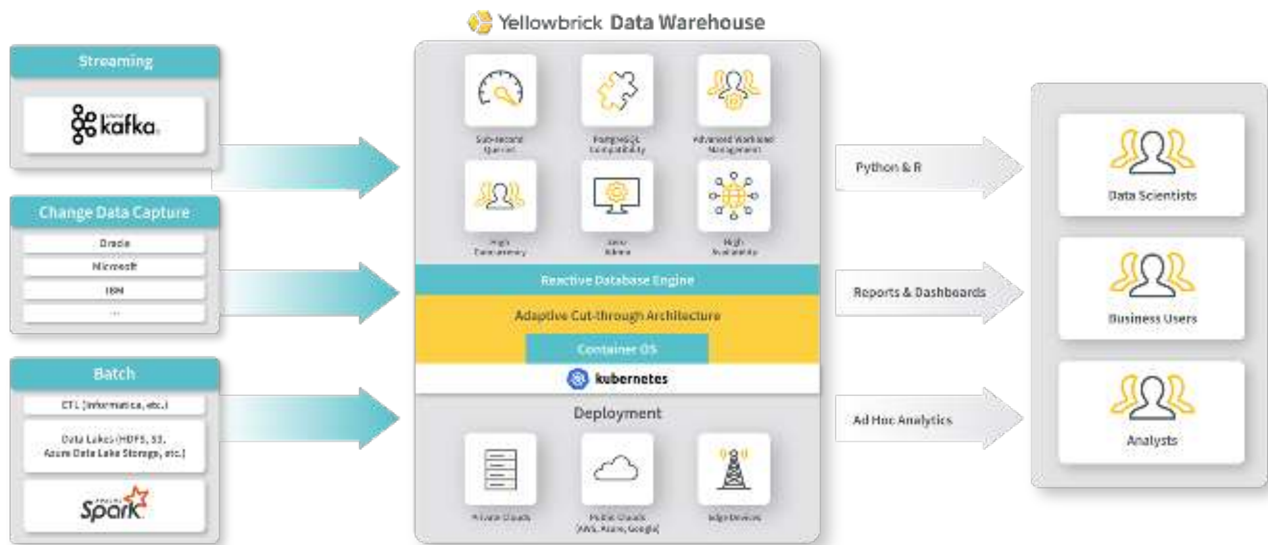

Figure 1. High-level view of Yellowbrick architecture

Evolution and impact of computer architecture

If you were to study database architecture and look at the algorithms in use across most databases built in the last 10-20 years, you would notice some standard assumptions, including:

- Databases 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.
- Linux provides excellent services for storage, network, and concurrency management and limitations.

At Yellowbrick, we've turned most of these assumptions on their heads. On modern instances using the latest hardware, did you know that:

- The maximum effective bandwidth of reading data from 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.
- Your CPU can run 10x faster if processing data from caches, rather than from main memory.

Yet, at the same time, technology gets in the way—and that technology includes today's operating systems, like Linux. On modern instances with Linux, did you know that:

- Built-in networking is 20x less efficient than creative alternatives.
- Built-in storage I/O stack and filesystems are 100x less efficient than creative alternatives.
- Built-in process and thread management is the enemy of high concurrency.
- Built-in memory management will frequently end up with the wrong memory in the wrong place.

These 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 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.

We've re-engineered the OS kernel and database from scratch, with all the above assumptions in mind, to optimize processing (the act of applying SQL operators to data structures), networking (moving data between nodes in a massively parallel cluster), and storage (storing and retrieving the data on high-performance media).

Inefficiency of high-throughput data processing with Linux

Today's hardware instances are routinely available with hundreds of gigabytes to terabytes of memory and dozens of CPU cores: At the time of this writing, a single off-the-shelf instance can support 2TB of RAM and 128 CPU cores (256 vCPU), and we envision that by 2023, 192 cores (384 vCPU) will be commonly available. Running generic software on these instances does not work well: Operating system schedulers were built to wait for events and "context switch:" Threads wait for events, such as a keypress, a network packet arriving, 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.

Thus, 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. Data access speeds in modern CPUs

Furthermore, 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 Joe
- Running a hash join of two tables for Mary

- Sorting data for Bill
- Waiting to look up a block in the filesystem for Steve 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, 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 Joe's join, then Mary's join, then Bill's sort, then Steve'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.

Introducing the Yellowbrick Kernel

To avoid these Linux-intrinsic problems, we built a new OS kernel from scratch. It implements a new execution model to eliminate measurable context switching overhead and penalties associated with accessing storage, the network, and other hardware devices. We do that with a new, reactive programming model for the entire data path.

The Yellowbrick Kernel is implemented as a "user space bypass kernel"—a Linux process that takes control of most of the machine and attached I/O devices. As a Linux process, it can run comfortably in container environments such as Kubernetes, in virtual machines, or on bare metal. It assesses how much "bypass" capability is possible in each environment and then adapts to use as much of it as it can, so it performs optimally in a VM on a 10-year-old laptop, a container in Amazon Elastic Kubernetes Service (EKS), on OpenShift in a private cloud, or on bare metal in a custom-designed blade server. When Yellowbrick starts, Linux is relegated to being a supervisor agent that collects logs and statistics, with all core data-path functionality bypassing it completely.

Some of the principles of this new programming model are described below.

Memory management

Yellowbrick intrinsically understands non-uniform memory architecture (NUMA) machines. At database startup, almost all memory in the system is handed to Yellowbrick and pinned (to make sure Linux never swaps in/out our process). Physical-to-virtual mappings are noted so hardware devices can directly and safely access the database memory bypassing the kernel.

Because databases run for a long time, issues such as cache locality and fragmentation can become a problem for memory allocators. Yellowbrick groups allocations by query lifetime to avoid fragmentation and provides allocation semantics for the runtime that can allocate commonly used data structures in just tens of CPU cycles.

Threading and processes

Yellowbrick has a new threading model based on reactive concepts such as futures and co-routines. 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 its own primitives for synchronization (Mutex, Semaphore, Condition Variable, and so on) 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 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 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 drivers

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, and so on) are present that fall back on the Linux kernel or on software emulation.

Networking

Like many modern microservices-based software stacks, Yellowbrick is implemented in a variety of different languages. We primarily use C, C++ , and Java, with a sprinkling of Go and Python where necessary, and these services need to talk to each other. Networks are unreliable in private and public clouds alike: Complex firewalls may make decisions to terminate connections, public cloud data center staff may choose to occasionally pull cables, and sometimes optical transceivers and cables fail.

Several different transports are both in use and under active development for YBRPC:

- 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. The CPU doesn't have to touch or copy the data; both send-side or receive-side and latency is measured in nanoseconds. This is particularly important for the MPP fast path used during query processing.
- RSocket over TCP: RSocket (Reactive Sockets) is a new emerging network stack used by modern microservices applications. It provides connection multiplexing, load balancing, and transparent reconnection in the event of network failure. Yellowbrick uses RSocket over TCP for reliable and efficient communication between different parts of the stack, but not in the MPP fast path.
- Amazon Scalable Reliable Datagrams (SRD) is technology only available in AWS with Elastic Fabric Adapters. It provides a fast, efficient kernel bypass for moving data across Amazon networks without having to use the Linux kernel, TCP, or UDP.
- Linux TCP and/or UDP: We have implemented a transport layer for generic kernel-based TCP. This is used when other more efficient transports are not available. We are implementing a custom UDP transport with DPDK user-space bypass technology to further improve database networking performance when running on generic cloud infrastructure.
- Unix Domain Socket: When processes are known to execute on the same node, this is more efficient than using a TCP socket.

The use of abstracted, high-performance, zero-copy networking with standard interfaces brings benefits to the Yellowbrick database that can't be matched by legacy databases: We 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. Our networking technology 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.

By supporting transparent reconnection, the database is also more robust to the network disconnects that happen from time to time.

Cluster parity filesystem

To lower costs while improving availability and reliability, Yellowbrick implements 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 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 do 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 in a manner similar to RAID-5 or RAID-6 in a disk-drive configuration. In RAID, blocks are rotated across several disk drives, with reconstruction data added as data is written. The simplest approach is RAID-5, which reconstructs data using XOR: If one drive in a set fails, its data can be reconstructed by XOR’ing together the remaining data, but if two drives in a set fail, data is lost. RAID-6 extends RAID-5 with an additional mathematically computed code such that two drives out of a set can fail without data loss.

Yellowbrick implements file-level erasure coding with ParityFS. 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 uses 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, those production customers that have rarely experienced node failures never even noticed. This technology, along with its TCO benefits, performance benefits, and integration into MPP database processing, is unique to Yellowbrick.

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, Execution Engine, Workload Manager, and Query Compiler. There's also a fifth, optional hardware accelerator (the Kalidah processor) for customers using our Andromeda optimized instance in private clouds, which offloads lots of computation.

Hybrid 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. 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, not scale-out, storage engine. It's optimized for low commit latency for real-time streams, such as those from Kafka or CDC tools.[1] When streaming data (such as all numbers of `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. How many real-time, streaming transactions per second Yellowbrick can do from sources such as Kafka or CDC is thus a function of commit latency, and the fastest path to the lowest possible commit latency is to avoid networking and commit immediately to disk.

The Yellowbrick row store is 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 runs as part of the main database front-end service, which is a regular Linux process and does not use the Yellowbrick Kernel (see "PostgreSQL compatibility" section). It's stored on mirrored, highly available storage volumes: For AWS instances, on EBS volumes; for other cloud vendors, on similar storage volumes; for on-premises instances, on four-way replicated SSDs. We don't erasure-code the row store because doing so would significantly affect commit latency and the volume of data is kept small.

---

[1] When using such tools, very-high-bandwidth applications can choose to micro-batch the data by accumulating, say, 30 seconds worth and committing it in one shot, but use cases that require truly recent data must insert events as they happen.

When the row store grows to the size where it will have a measurable impact on query performance, it is flushed in the background to the column-oriented store. Users and admins need not concern themselves with this flushing process.

We have implemented 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 column store without writing intermediate data to disk. Upon commit, the data will be persisted to the columns store instead.

Column-oriented store

The column-oriented store (column store) 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.

In Yellowbrick, we've been creative about how our column-store indexes and finds data efficiently on SSD media and cloud storage. The column store runs entirely within the Yellowbrick Kernel framework and stores data on BBFS in erasure-coded files to save space and reduce cost. Static partitioning is not required to achieve acceptable performance: As tabular data is written, rows are distributed (typically according to the hash of a column value) or replicated 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 together, 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 4KB-to-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 we've designed for SSD storage, which excels at large numbers of random disk operations (IOPS). When we can execute tens of millions of IOPS while consuming a minimum amount of CPU, we can afford to turn table scans into massive 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 we need to be careful to keep queue depths low to ensure we don't use too many L3 cache pages. Achieving massive bandwidth with tiny queue depths requires an extraordinarily efficient IO framework.

Because statistics (which include big data structures such as HyperLogLogs) are stored along with every shard, the act of updating database statistics becomes something that DBAs don't have to worry about. It can be completed very efficiently just by combining summary information from the shards in the system. The process is so fast it's done automatically in the background, so unlike traditional databases, there's no need for admins to keep statistics up to date.

Yellowbrick deletes rows in the column store 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. However, we didn't set out to build an OLTP database, so this trade-off is acceptable.

Yellowbrick has a built-in garbage collector that will periodically sweep up fragmented shards, remove deleted rows, and recombine them efficiently. It does this incrementally in very small units of storage, such that unlike older databases where performance suffers greatly during "vacuuming", in Yellowbrick, 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 & 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, rather than at the row level as in a transactional database. 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 column store can run concurrently with bulk loads.
- Locks are acquired for queries early in the processing workflow. (See the "Query processing" section for more information.)

Hybrid Execution Engine

The Execution Engine (EE) is as much a reason for Yellowbrick's speed as the Kernel and the Storage Engine. Just as our database stores data in both rows and columns, we also execute queries in a row-oriented or column-oriented fashion. 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, we'd build a query plan that 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") and more modern ones that push data upward, 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.

A side effect of this approach to flow control is that 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 whatsoever. Flow control guarantees optimal use of memory and keeps data cache-resident wherever possible.

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.

The EE doesn't contain any code to implement SQL operators, only templates for constructing them and the frameworks for gathering execution statistics, spilling, cancellation, and flow control. When a query is loaded, the operator nodes are instantiated via compiled LLVM code and then wired up by constructing the appropriate links. The EE supports different subclasses of links depending on how we wish data packets to be transmitted as well as implementations suitable for debugging, error injection, and performance testing.

All the above is implemented using the reactive, asynchronous, non-blocking framework of the Yellowbrick Kernel to ensure maximum use of all possible resources.

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.

Yellowbrick SQL operators can choose the type of packet format on which they are able to work, and Transpose operators are injected into the query plan to optimally rearrange data accordingly. For

example, data is transposed from columns to rows when moving between a Scan operator and Join operator. Data is transposed to columns from rows when tabular data is written to disk.

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. As many as four partition columns can be chosen per table.

Most MPP databases implement partition pruning to find data more rapidly by subsetting the data that needs to be scanned. In Yellowbrick, shard indexing is so efficient, there's no need for pruning to find or delete data. Instead, all knowledge of partitions is pushed down to the executor itself. Admins configure Yellowbrick to use partitioning solely to reduce memory usage for massive joins or aggregates. Consider a couple examples:

1. A hash join of two massive tables with perhaps hundreds of billions of rows (a fact-to-fact join). The traditional 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.
2. 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 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, we can execute the join a partition at a time because we know the data in the partitions is non-overlapping. If we chose 1,000 partitions, we would do 1,000 smaller joins rather than one massive join, thus 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 we know the CDRs have been partitioned by hour, we can divide and conquer the query by running one aggregate per hour and incrementally emitting 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.

Operators

In Yellowbrick, algorithms and data structures used by key operators such as scan, join, aggregate, and sort have been implemented rather differently from traditional databases to scale sufficiently well and not suffer strange execution artifacts present on high core-count CPUs, such as cache aliasing.

Yellowbrick implements a dynamic set of operators that can adapt their strategy at runtime rather than assume perfect strategy selection by the query planner ahead of time. Earlier in the evolution of Yellowbrick, our planner estimates were rather inaccurate, so we were unable to decide how much memory to allocate a given operator or if it would require spilling or not. Instead, we made our operators adapt: At runtime, for example, the aggregate operator can change its strategy from hash-based to merge-based (by sorting data). The join operator can choose between different hash representations depending on the size of the data structure relative to the CPU's cache.

Yellowbrick does not use a transparent buffer pool like that used in traditional databases. Operators are built primarily using an in-memory execution paradigm, just like high-performance, in-memory databases such as SAP HANA. They assume all data can be resident and fit in the allocated memory. If this isn't the case, a memory-usage monitor will instruct operators to start spilling partitions to disk. Data can be spilled incredibly quickly—the write rate of SSDs is of the order of many GB/sec or more per node—and is typically spilled by writing row packet data structures directly to BBFS scratch files. The algorithms used by spilling versions of operators differ from those used by memory-resident ones and are designed to maximize SSD bandwidth, as well as control media wear.

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 in order to enable parameterized queries to still 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. We also create `BETWEEN` predicates from the minimum and maximum values

present in the build sides, 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 whatsoever.

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 our 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.

Type optimization

Yellowbrick implements most standard SQL data types—at the time of this writing, these include variously-sized integers, floats and doubles, decimal numbers, fixed and variable-length characters, and so on—as well as a few special types such as IP addresses and UUIDs. Lime tries to avoid any form of runtime type inference, instead aggressively (as possible) implementing strong, explicit types ahead of time for every expression operator and value or parameter, in every expression, in every data packet, in every SQL operator, for the entire query from top to bottom.

This typing process contains several optimizations and phases:

- Tracking and propagating the possible nullability of values: Operators that don't need to check for null involve far fewer machine instructions than those that do.
- Tracking and propagating the maximum lengths of variable-length data: Being able to make optimal memory and stack allocations for all values, even in and around string processing functions, increases cache efficiency and reduces memory used.
- Tracking and propagating the precision and scale of decimal numeric values: In as many cases as possible, Lime will generate optimized code with hard-wired precision and scale values for inlining.
- Tracking and propagating the ability to vectorize operators: In cases where SIMD operations can be used to operate on multiple values in one shot, Lime will do so.

All of these typing processes help the system generate more-optimal machine code.

LLVM compilation

Lime uses 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: We don't just compile hotspots or compile the "core loops" of the operators; the code implementing the entire operator is ultimately expanded to one set of code that can be aggressively optimized by the LLVM optimization passes. Which optimization passes are chosen is a function of how much CPU time we estimate will be consumed.

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. The code from the resulting segments is then linked together to form the query that's dynamically loaded and executed by the Yellowbrick Kernel. This parallelization allows simple queries to be compiled and executed in tens of milliseconds, while very large, complex queries can still compile within a second or two, 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 quite likely has the fastest database regular expression implementation ever created, although we've not yet formally benchmarked it.

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, we'd rather save milliseconds of compile time for short, tactical queries or 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 Yellowbrick Kernel, versions of libraries, the query plan in use, the statistics of the query, and so on. Yellowbrick factors all these dependencies to try to either return a query's object code directly from its plan, or, if necessary, return object code or parts of object code for query fragments passed to the LLVM compiler infrastructure.

Kalidah processor

The Kalidah processor is new accelerator intellectual property developed by Yellowbrick for our Andromeda optimized instance. Designed for implementation in FPGA or ASIC, it accelerates bandwidth-oriented data processing tasks used during table scans such as data validation, decompression, filtering, and compaction. Kalidah supports all data types currently supported by the Yellowbrick database, including decimal numbers and variable-length strings.

For more details about Kalidah, see our "Andromeda Optimized Instances" white paper.

PostgreSQL compatibility and query planning

When our founding engineering team started, they initiated building the Yellowbrick database from the bottom up through extensive prototyping. They spent months prototyping and experimenting with different code patterns to achieve the highest level of data-crunching performance while using as little memory as possible. They tried different hardware access primitives, CPU instructions and architectures, and memory-access patterns, using several example queries from the TPC-H and TPC-DS test suites. Their goal was to build the smartest-possible, human-implemented, bespoke assembly-language implementations of the code behind these SQL queries. When they felt the code was optimal, they set about building the full runtime kernel and query compiler to formally implement their findings.

We decided early in our history to use open source technology for external access to the database. PostgreSQL was 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 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 typically is 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 in the future we will ship custom, higher performing *DBC drivers while continuing to maintain Postgres compatibility. PostgreSQL's metadata catalogues 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 only for PostgreSQL compatibility and internally aliases to `VARCHAR(64000)`.

We've extended PostgreSQL's SQL dialect and functions for Oracle, Microsoft, and Teradata compatibility, and added 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 not supported by Yellowbrick at this time.

We've added numerous utility commands to the SQL grammar for tasks such as managing the workload management subsystem 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 extensively
- Scan selectivity algorithm replacement
- Support for various forms of correlated sub queries
- Support for using and combining incremental statistics using big data algorithms
- Query auto-parameterization to support plan re-use
- Partitioning: Support for 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 painlessly
- 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.

We have trained our query planner to try to avoid writing intermediate data structures to disk wherever possible. Flash memory prefers to be read rather than written and we seek to maximize its lifetime: Considering the performance of scans and SQL operators in Yellowbrick, it's often as efficient to run a sub-query several times as it is to write an intermediate, temporary result to disk. Outside these areas,

we have made very few innovations in the area of query planning; old and established commercial databases have excellent query planners, and we strive to do as good a job as they do. Our query planner has a dedicated development team with its own implementation roadmap, and it increases in maturity with each new release.

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 higher priority than anything else going on in the system.
- "Bad actor" rogue queries, perhaps written by users with a 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. (See next page.) The meanings of these stages are generally self-explanatory.

When a query is submitted, we know little 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 of information is known about it: 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 cancelled or move into an error condition at any point in the flow.

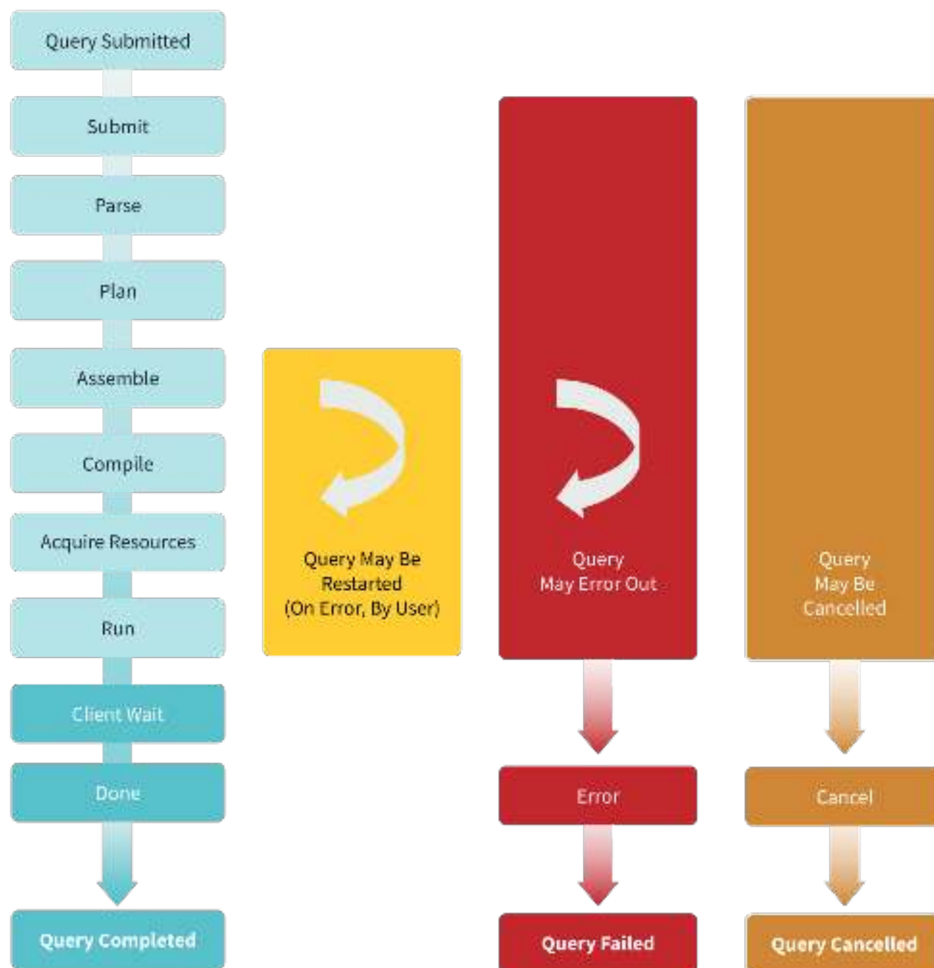

Figure 2. Query execution in Yellowbrick

Monitoring and introspection

The Workload Manager is the single source of truth for everything currently going on 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.

Resource management

Yellowbrick can proactively allocate and manage the following system resources:

- Persistent storage space—through disk quotas
- Spill space
- Memory
- CPU—through priorities

We divide available resources into pools. Queries are routed to a pool by rules executing within the Workload Manager (see below). We may have, 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 is able to support, which may be fixed or flexible: Fixed concurrency is useful when we know 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.

Control points and 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, if we want 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 single-row retrievals.

Writing rules

Workload Manager rules are written in JavaScript, providing incredible flexibility. Properties about the system or currently executing queries 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, we have created 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

We have many different types of rules that 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 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 allow 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 runtime 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 systems-management tools. This is how to 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 cancelled, or erroring out.

Logging capabilities are available to trace rule execution. Rules have priorities and are evaluated in priority order. A container of rules is called a *profile;* profiles may be imported and exported to/from JSON to move them between instances.

Only one profile is active in the database at a given time, but the profile 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.

Business continuity

Yellowbrick has been designed from Day 1 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. These design imperatives combine to give Yellowbrick the native, best-in-class functionality that ensures high availability. No third-party tools, services, or "enterprise editions" are needed.

Yellowbrick is designed for data warehousing and analytics rather than high-end online transaction processing, so we did not consider "five nines" of availability—99.999% uptime—a requirement. Five nines allow only about five minutes of downtime per year and necessitates no maintenance windows, fully online upgrades, and the like. Such incredible uptime comes with design complexity and runtime performance implications that we did not consider appropriate. Instead, our architecture is centered around achieving "three-and-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. We don't claim to have mainframe-class or Oracle-class stability and availability, but we are proud of our track record: Yellowbrick routinely backs production, online, ad hoc 24/7/365 business-critical financial applications for many Fortune 500 customers with minimal unscheduled downtime.

Due to the massive amounts of data growth experienced by customers, expanding the cluster for more storage or compute capacity is an online activity that does not require scheduled maintenance windows.

High availability

Yellowbrick is clustered software, running on several different server nodes. Instance redundancy has many different aspects:

- Fans: If a fan rotor fails, do the remaining fans have enough suction to cool the server?
- Power supplies: If a power supply fails, do the remaining power supplies deliver sufficient power to power the server?
- Power: Is the power supplied to the server from multiple phases, so that if one phase is lost, the server remains powered up?
- Network connections: Are the servers connected to the network through multiple interfaces and cables?
- Switches: Does the network itself have more than one switch, with the instances connected to both switches so that the database can survive switch failure?

Our Tinman and Andromeda instances sport all these redundancies, making the database hardware itself "highly available." Failures in any of the above components will not result in unscheduled downtime or cluster reconfiguration changes (more on that below).

For more details about our optional hardware instances, see our "Andromeda Optimized Instances" white paper.

ParityFS: Data protection against node failure

Where cluster nodes have persistent storage and the cluster is highly available, our 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 minimal and typically unnoticed; failure of a node or SSD results in a cluster reconfiguration event which causes momentary service degradation.

Cloud Mirror: Data protection for cloud instances

For instances running on generic container infrastructure such as public clouds, the clusters do not provide guaranteed high availability. Multi-phase power may not be available, and the level of redundancy in connections to the network is unknown. Cloud nodes running within a given "availability zone" can fail concurrently and *en masse*, so a different strategy is needed.

Non-HA instances, or those with only ephemeral storage such as affordable public cloud instances (persistent cloud storage such as EBS is prohibitively expensive and low on performance) use technology we call Cloud Mirror. In this approach, files are written to local, perhaps ephemeral, storage directly via the BBFS filesystem, without using ParityFS. They are also synchronously streamed to cloud object storage.

Transactions are not committed until they have been written in both locations. When a file is read, if missing from BBFS it will be paged in from cloud object storage. The file can be loaded incrementally with missing blocks, rather than in one shot, to improve response times and retrieve only the data necessary for the query.

Conversely, if the local BBFS filesystem is full, we will remove older data blocks to make space for newer data. This allows smaller instances to serve as a cache for a potentially much larger object store.

Expansion, fault, and reconfiguration tolerance

Cluster reconfiguration events happen when changes are made to the topology of the database cluster. This can happen due to several physical changes such as expanding the cluster by adding nodes or dealing with node failure and drive failure. Cluster reconfiguration causes brief degradation for users due to a temporary pause in query processing that lasts for a few seconds to at most a minute (larger systems, with more nodes and more storage, take longer to reconfigure). The Workload Manager will automatically pause and restart most queries: those that haven't returned data to a user, and those that don't involve certain types of write or load transactions. However, queries that have started

returning data to the users, bulk loads/unloads, and certain classes of write transactions, will be aborted and the user will receive an error, requiring resubmission.

If new nodes have been added to a cluster, redistribution of data on the new nodes will take place in the background with little impact to query processing, so capacity expansion is viewed as an online activity despite the brief reconfiguration pause.

Yellowbrick has been designed to automatically restart as quickly as possible in the rare event of a crash. Crashes during query execution typically result in a cluster reconfiguration event as described above. Crashes during query parsing and planning will normally just terminate the session of the user that suffered the crash, leaving all other user sessions intact without disruption.

Backup & 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 catalogue 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 our roadmap, we have committed to exposing the snapshot functionality and supporting “time travel” queries for general SQL use.

SQL-native backup and restore

Backup and restore operations are implemented as SQL operations, and as such, leverage the incredible performance of the Yellowbrick Kernel, Execution Engine, and Storage Engine. By doing 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 formed into a proprietary backup format and 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 backup 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 & 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. This 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 catalogue, 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 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 as desired. No third-party utilities are required for replication. Both DDL and data are replicated. Replication takes place over SSL-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 a 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.

In Yellowbrick, failover is not automated because it is considered a rare event: Individual instances are highly available and protected, so failover should only be necessary in the event of a mass loss of connectivity or a real 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 a 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 warehouses need to be able to ingest large amounts of data rapidly and spit it back out again. Sometimes data ingest and extraction need 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, whereas other times it's initiated as part of a complex scripted pipeline. Yellowbrick caters to most of these scenarios.

Yellowbrick supports a bulk data path that's separate from the normal streaming data paths. However, the streaming data path can run in bulk mode when needed. We explore what this means below.

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 in nature 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 Execution Engine and Storage Engine. These operations typically are bound by the speed of the network and the number of CPU cores in use on the sender or receiver: We've seen rates of around 10TB/hr on relatively fast networks, and on high-speed switched fabrics, we expect to be able to go even faster.

The bulk transfer protocol proxies external Yellowbrick tools that securely negotiate and open parallel TCP/IP connections to each node. The protocol was designed for high-performance transfer across multi-gigabit enterprise network topologies, including the proxies and NATs commonly used by public cloud provider (PrivateLink) gateways.

You can initiate bulk load and unload through SQL or by using traditional client tools. (See Table 2.)

Use Case	Tools	Description
Load & unload	ybload and ybunload	Load/unload database data from/to files or cloud object storage
Backup & restore	ybbackup and ybbrestore	Online full, cumulative, and incremental backup of database data. Hot-standby full restore
Data lake integration	ybrelay	Spark relay for Avro, Parquet, and other data formats

Table 2. Yellowbrick client tools

You can move data from/to local filesystems, NFS filesystems, or cloud object stores such as S3 and MinIO using various credentials. The client tools are written in highly tuned but portable Java and are in production use by our customers on all sorts of platforms (AIX, Windows, Apple MacOS, and Linux). Through SQL, a full utility grammar is provided to configure remote storage locations and file formats, list files, peek files, and initiate loads and unloads alongside a traditional external table interface to allow interactive querying of data stored externally.

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, and the list is growing to include common Big Data formats such as Apache Parquet as well as traditional delimited data and BCP files. Yellowbrick Manager also enables browsing and loading of data through a web browser.

A standard Java interface is available for the data loading library to allow direct programmatic integrations by ETL, CDC, and other data movement tools. The load path also includes support for bulk updates, bulk deletes, and "upserts."

Some traditional enterprise integrations like to move data from and to the database in a client/server fashion. For that purpose, we provide a set of Java-based client tools for hybrid-cloud deployments; they support data transfer across public cloud provider (PrivateLink) gateways and cloud object storage standards including AWS S3 and Azure Blob Storage. Third-party partners are also integrating Yellowbrick tools into their own product offerings to enable high-performance integrations.

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 row-oriented store, we've observed rates of around 3 million rows/sec. 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 warehouses that struggle to do streaming ingest efficiently, the hybrid nature of the Yellowbrick Storage Engine means we can 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.

Concurrency

Yellowbrick can efficiently query tables as data is being written to them. Large bulk loads 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 of our customers have tables measured in hundreds of terabytes in which 30 percent of the data is changing daily, updated once a minute.)

Pre-packaged integrations

Along with all the ecosystem partners supported by Yellowbrick, we also supply off-the-shelf integrations with Oracle Golden Gate, Kafka, and Spark. These integrations are configurable to run in bulk mode or streaming mode, depending on the volume of data being generated versus the recency requirements.

Security, systems management, and monitoring

Enterprise-class databases must be secure, and easy to manage, monitor, and configure. We've built a variety of mechanisms for that into the product 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 for a world where we assume everything is "private by default." No public buckets are enabled by default to gain access to data, no built-in guest users are present, and strict access must be granted to all data and management functionality.

Authentication

Database users can be authenticated locally or with LDAP and Active Directory. We are currently working on deep integration of OpenID Connect and SAML 2.0 multi-factor authentication (MFA) for cloud-native products (including Azure Active Directory, Okta, and Ping).

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 our database is based on PostgreSQL, we deliberately split the privileges afforded to the super user 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

Access to all database schema objects is fully role-based. Below the granularity of a table, Yellowbrick allows granting access to columns of a table. PostgreSQL administrators will be familiar with this concept.

Encryption of data at rest

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

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 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 does 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.

TLS support

TLS 1.2 is supported 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 cross-database replication sessions.

Protegrity partnership

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

Systems management and monitoring

We've already covered instrumentation and statistics gathering in the database, and its visibility through system views. However, the database itself needs managing and instrumenting from the outside. In the future, we plan to integrate a fully query-able configuration management database (CMDB) into Yellowbrick Manager to present a unified, extensible view of everything outside the core database. In the meantime, the following functionality is available.

- **Alerting and SNMP integration**
  Numerous alerts can be configured based on either state changes in the database system—changes in cluster state due to hardware failures, changes in network connectivity and so on—or on the threshold of various inbuilt monitors such as storage space used, connections used, or integration status changes with third-party systems. The alerts can send SNMP traps to integrate with systems management solutions, send emails or post to a web service (HTTP). For things going on inside the database, the Workload Manager can programmatically send SNMP traps or emails or communicate status to HTTP endpoints.

- **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 in the public cloud. By default, Yellowbrick maintains remote, unidirectional connectivity to Yellowbrick's internal SaaS monitoring platform. Any crash mini-dumps, key telemetry, and key logs are sent back over this phone-home connection to enable our Customer Success team to monitor customer systems 24/7. In the event of any issues, Yellowbrick will know about them first. Optionally, 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 the feature completely.

- **System logs**
  Yellowbrick supports a remote syslog capability to allow all systems logs to be forwarded to a log collector of choice. This typically is unnecessary because Yellowbrick provides all database support. However, for some customers running “dark” networks in private data centers that are disconnected from the internet and have limited staff access requirements, all logging data can be forwarded to a centralized aggregator for archive, viewing, and searching.

Summary

This white paper describes how modern computer architecture requires substantial change and innovation in the software stack to truly take advantage of the promised gains in performance, efficiency, and economics, and to address the challenges of distributed and real-time data.

For data warehouses, several assumptions about how software is built need to be revisited. Yellowbrick has set the bar higher than any other company in the industry with respect to maximizing the value of today's systems, whether they're located in public clouds or private data centers.

We've augmented this substantial innovation in database storage and execution with enterprise-level workload management, query planning, business continuity, and high-throughput parallel data movement.

The result is a new kind of data warehouse that provides the best economics in industry, along with all other expected features and functions of a mature product that can be trusted to help run your business faster and more efficiently.

Yellowbrick Data Inc.
250 Cambridge Ave Ste. 300
Palo Alto, CA 94306
yellowbrick.com

© Yellowbrick Data, Inc. All rights reserved. Yellowbrick Data and the Yellowbrick logo are trademarks of Yellowbrick Data. All other trademarks are the property of their respective owners. Information is subject to change without notice.