Applying Control Theory to Create a Self-Optimizing Database

The Technology Underpinnings of ScyllaDB’s Autonomous NoSQL Database
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTRODUCTION: THE PATH FROM AUTOMATION TO SELF-OPTIMIZATION</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>STEERING THE SHIP: FEEDBACK AND CONTROL SYSTEMS</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>SCYLLA’S SELF-OPTIMIZING SCHEDULERS</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>BANDWIDTH, DISK I/O, CPU</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>COMPACTION</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>NODE RECOVERY</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>CONCLUSION</strong></td>
<td>8</td>
</tr>
</tbody>
</table>
INTRODUCTION: THE PATH FROM AUTOMATION TO SELF-OPTIMIZATION

Machines are becoming less and less dependent on human operators to accomplish their tasks. We see it every day: self-driving cars, industrial robots, and customer service bots. Culturally, humans are increasingly, though not yet entirely, comfortable delegating control to machines.

In parallel, IT organizations have sought to automate the data center, from infrastructure provisioning and management to application delivery. The emergence of technologies such as Mesosphere and Kubernetes have made the data center more programmable, and hence automatable, than ever before. Automation, however, is just a step on the path to autonomous infrastructure. While automation enhances repeatability and lowers the risk of operator error, today’s implementations are still too-often static and linear. An automated system merely performs the same tasks over and over. An autonomous system, in contrast, is self-optimizing. Such a system adapts to changing conditions without operator intervention.

Databases have a checkered history of claims surrounding self-optimizing capabilities. They are also sometimes referred to as having ‘autonomic’ or ‘autonomous’ capabilities. Some vendors confusingly position ‘self-driving’ databases as fully self-contained and self-regulating black boxes designed to eliminate Database Administrators (DBAs). The best known example of this approach is Oracle’s Autonomous Database Cloud, which claims to deliver automated patching, upgrades, and tuning, along with routine database maintenance tasks. On close examination, these capabilities are indistinguishable from those of managed cloud databases. Using these criteria, Amazon, for example, could just as easily tag Amazon DynamoDB as ‘autonomous’, though it clearly is not.

To avoid this confusion, we must distinguish between third-party cloud vendor capabilities and those that are intrinsic to the infrastructure itself. A self-regulating database is vastly more important when internal IT teams are responsible for the performance and stability of the system. The real challenge for database management systems is to offer self-optimizing capabilities to any DBA, whether on-premises, private cloud, or public cloud deployments are involved. As such, a self-optimizing database should augment, not eliminate, the DBA function.

A self-optimizing database needs to navigate between the extremes of too much and too little control. Too much control enforces endless configuration and byzantine tuning parameters that place the burden of performance and stability on the operator. Too little control puts operators at the mercy of external forces and vendor support teams. The trick is to strike the appropriate balance.

STEERING THE SHIP: FEEDBACK AND CONTROL SYSTEMS

The solution to this conundrum can be found in systems theory. The founding metaphor behind systems theory is Kubernetes; not the familiar Kubernetes that we mentioned above! ‘Kubernetes’ is the Greek word for helmsman, and gives us the term cybernetics. A helmsman is an apt image for a self-regulating and self-optimizing system. The helmsman guides the boat toward a compass heading by continuously adjusting the rudder and sheets, while responding dynamically to multiple sources of feedback, such as the direction of the wind, the strength of the current, the angle of the waves, and the stars.

To follow this metaphor, databases are complex systems that can use feedback to continuously self-regulate and self-correct with changing conditions, such as spiky workloads and shifting deployment topologies. Control theory can be used to strategically automate key aspects of
runtime database administration, providing the tools needed to achieve optimal operational overhead, while eliminating routine yet risky administrative tasks. This approach treats the database as a closed signaling loop, in which actions generate changes in the environment, which in turn trigger changes to the system in a recursive fashion.

Closed-loop control systems traditionally involve control processes and actuators that progress output to a desired state. The gap between the current state and the desired state is referred to as the error. The error can be returned to inputs as feedback. (That’s why closed-loop control systems are also sometimes called feedback control systems.)

Industrial controllers provide a real-world example of the way that control theory can be applied to technology. Take, for example, an industrial controller that regulates the water level in a holding tank by itself, without human intervention. To maintain the water at a prescribed level, a valve is designed to function as an actuator. Using feedback from the system, the controller opens and closes the valve to maintain a consistent water level. Figure 1 illustrates this example.

Like the storage capacity in a holding tank, database resources are finite. Internal database operations compete with customer-facing workloads for shared resources: CPU, disk I/O, and network bandwidth. As any DBA can attest, traditional databases are static and brittle in the way they handle this interplay. Operators need to fiddle with various database-specific tuning parameters on each server. There’s very little predictability, and what might be the ideal balance today may no longer suffice tomorrow. History has shown that that kind of centralized decision-making can be disastrous in complex systems. Just as the actuator that controls the valve, a database regulates resources available to different workloads. A self-optimizing database can remove the operator from the loop and automatically tune the database to achieve specific performance objectives with minimal planning.

**SCYLLA’S SELF-OPTIMIZING SCHEDULERS**

To achieve self-optimization across database operations, Scylla embeds control processes as *schedulers*. Scylla’s schedulers behave as actuators in a control system. Like the valve in the holding tank discussed above, Scylla schedulers enable operators to strategically allocate resources, without getting lost in the minutiae of tuning settings. By doing so, they are able to uphold the ratio of resources that operators devise, automatically increasing or decreasing the rate at which a process is executed as workloads ebb and flow.

To achieve self-optimization, Scylla embeds several scheduler/actuators, each of which is responsible for resources such as network bandwidth, disk I/O, and CPU.

**BANDWIDTH, DISK I/O, CPU**

Databases have long struggled with a conundrum: exponential growth in storage volume has been accompanied by linear growth in disk I/O bandwidth. As infrastructure has become capable of storing ever more data,
the associated networks (internal to the chips) have struggled to keep pace. The results are unpredictable performance and bottlenecks.

In a distributed database, many actors compete for available disk I/O bandwidth. Databases like Cassandra, Redis, and HBase control I/O submission by statically capping background operations such as compaction, streaming and repair. Getting that cap right requires painstaking, trial-and-error tuning combined with an expert knowledge of specific database internals. Spiky workloads, with wide variance between reads and writes, or unpredictable end user demand, are risky. Set the cap too high and foreground operations will be starved of bandwidth, creating erratic latency and bad customer experiences. Set it too low and it may take days to stream data between nodes, making auto-scaling a nightmare. What works today may be disastrous tomorrow.

Scylla’s goal is to autonomously maintain balance between the two ideals of simultaneously maintaining system stability while preserving customer-facing Service Level Agreements (SLAs). Scylla tackles this challenge by managing requests inside the database itself. Most databases delegate requests to lower-level kernel space queues. Queueing requests in user space does not improve latency by itself; it merely swaps one queue for another. However, by queuing requests in user space, Scylla gains control over processing of I/O requests. For instance, Scylla can:

• Provide metrics that enable application-level throttling
• Prioritize requests and process them selectively
• Cancel requests before they impact a lower layer’s software stack.

The diagram below shows this architecture at a high level. In Scylla, requests bypass the kernel for processing and are sent directly to Scylla’s user space disk I/O scheduler. The I/O scheduler applies rich processing to simultaneously maintain system stability and meet SLAs.

Figure 2: On the left, requests generated by the userspace process are thrown directly into the kernel and lower layers. On the right, Scylla’s disk I/O scheduler intermediates requests. Scylla classifies requests into semantically meaningful classes (A and B), tracks and prioritizes them, guaranteeing balancing while ensuring that lower layers are never overloaded.

COMPACTION

One key example of a routine maintenance task that benefits from self-optimization is compaction. Compaction is a process that is unique to NoSQL databases with storage layers based on Log-Structured Merge-trees (as opposed to B-trees). The list of databases that employ compaction includes Google BigTable, HBase, Apache Cassandra, MongoDB and Scylla. In these systems, data is written into memory structures (usually called memtables) and later flushed to immutable files (called sorted string tables, or SSTables).

Over time, these tables accumulate and at some point must be merged together by the process known as compaction. This background task occurs occasionally and fairly unpredictably, and often requires significant bandwidth and CPU cycles to do their work. As such, compaction always presents the risk of cascading effects that can cripple a running deployment.
Configuring compaction manually requires intimate knowledge of both expected end user consumption patterns along with low-level database-specific configuration parameters. For example, DataStax’s documentation for Cassandra reveals a number of fairly obscure configuration settings used to tune compaction. A small sample includes:

- `concurrent_compactors`
- `compaction_throughput_mb_per_sec`

Scylla’s self-optimizing operations eliminate such low-level settings. Instead, they employ algorithms to let the system self-correct and find the optimal compaction rate under varying loads. This radically lowers the risk of catastrophic failures, and also improves the interplay between maintenance and customer-facing latency.

The impact can be seen in the following performance graphs.

![Figure 3: Throughput of a CPU in the system (green), versus percentage of CPU time used by compactions (yellow). In the beginning, there are no compactions. As time progresses the system reaches steady state as the throughput steadily drops.](image)

![Figure 4: Disk space assigned to a particular CPU in the system (yellow) versus shares assigned to compaction (green). Shares are proportional to the backlog, which at some point will reach steady state.](image)
NODE RECOVERY

No database cluster is immune from node failure. A common operator’s task is node restoration. The challenge for node recovery is to introduce new nodes as quickly as possible, with minimal impact to production applications. When a fresh node is added to a cluster, there is a delay as the new node is populated with data. This is commonly referred to as the the ‘cold node’ problem. Cold nodes slow down the whole cluster. The key is to minimize that delay and have the node begin serving requests as quickly as possible.

Scylla takes a self-optimizing approach that leverages patent-pending technology known as heat-weighted load balancing. This approach
intelligently routes requests across the cluster during recovery, relying on an algorithm to optimize the ratio between a node’s cache hit rate and the proportion of requests it receives.

Over time, the node serving as request coordinator sends a smaller proportion of reads to the existing nodes, while increasing the proportion of reads directed to the restored node. The feature enables operators to restore or add nodes to clusters efficiently, since rebooting individual nodes no longer affects the cluster’s read latency. Taking into account the gradual increase in availability by new nodes, heat-weighted load balancing ensures that node restarts don’t impact throughput or latency.

With Scylla’s self-optimizing, heat-weighted load balancing, rebooting individual nodes no longer affects the cluster’s overall latency.

CONCLUSION

Scylla was architected and implemented by engineers with deep-level knowledge of operating systems, distributed systems, complemented by an appreciation for the power of control theory. A foundational architectural principle, self-optimizing capabilities manifest themselves in every aspect of the Scylla database. As such, Scylla has truly been built from the ground-up to deliver self-optimizing capabilities that deliver a range of benefits, including:

• The ability to run operational and analytics workloads against unified infrastructure
• Higher levels of utilization, and the elimination of wasteful over provisioning
• Significantly lower administrative overhead
ABOUT SCYLLADB

Scylla is the real-time big data database. Fully compatible with Apache Cassandra, Scylla embraces a shared-nothing approach that increases throughput and storage capacity as much as 10X that of Cassandra. AppNexus, Samsung, Mogujie, CERN, Grab, Olacabs, Investing.com, Eniro, IBM’s Compose and many more leading companies have adopted Scylla to realize order-of-magnitude performance improvements and reduce hardware costs. ScyllaDB was founded by the team responsible for the KVM hypervisor and is backed by Bessemer Venture Partners, Innovation Endeavors, Wing Venture Capital, Qualcomm Ventures, TLV Partners, Magma Venture Partners, Western Digital Capital and Samsung Ventures. For more information: ScyllaDB.com