informatics

MDPI

Article

Automated Configuration of NoSQL Performance and Scalability Tactics for Data-Intensive Applications

Davy Preuveneers * and Wouter Joosen

imec—DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; wouter.joosen@cs.kuleuven.be
* Correspondence: davy.preuveneers@cs.kuleuven.be; Tel.: +32-16-327853

Received: 3 July 2020; Accepted: 4 August 2020; Published: 8 August 2020

Abstract: This paper presents the architecture, implementation and evaluation of a middleware support layer for NoSQL storage systems. Our middleware automatically selects performance and scalability tactics in terms of application specific workloads. Enterprises are turning to NoSQL storage technologies for their data-intensive computing and analytics applications. Comprehensive benchmarks of different Big Data platforms can help drive decisions which solutions to adopt. However, selecting the best performing technology, configuring the deployment for scalability and tuning parameters at runtime for an optimal service delivery remain challenging tasks, especially when application workloads evolve over time. Our middleware solves this problem at runtime by monitoring the data growth, changes in the read-write-query mix at run-time, as well as other system metrics that are indicative of sub-optimal performance. Our middleware employs supervised machine learning on historic and current monitoring information and corresponding configurations to select the best combinations of high-level tactics and adapt NoSQL systems to evolving workloads. This work has been driven by two real world case studies with different QoS requirements. The evaluation demonstrates that our middleware can adapt to unseen workloads of data-intensive applications, and automate the configuration of different families of NoSQL systems at runtime to optimize the performance and scalability of such applications.

Keywords: resource optimization; hyperparameter tuning; machine learning; smart environments

1. Introduction

The amount and diversity of data that is being created, collected, stored and processed in enterprises is growing at an unprecedented pace. However, sophisticated data processing demands for predictable performance guarantees, as a potential degradation in performance introduces a risk where the time needed to complete the processing and analysis exceeds the window of opportunity to exploit the results by the application. This is for example the case in an online payment processing application that processes historical and contextual data to improve risk assessment without introducing unacceptable delays for customers and merchants.

Application architects are adopting NoSQL storage technologies that have been designed to provide good horizontal scalability for data-intensive computing and analytics applications. Not only are architects faced with the curse of choosing the technology [1] that best matches their data model, consistency and availability requirements, yet it is far from trivial to compare the relative impact and trade-offs between different performance and scalability tactics that must be applied when configuring the chosen storage technology [2,3]. Standard benchmarks can help evaluators in testing different storage systems and in facilitating performance comparisons. For example, the Yahoo! Cloud Serving Benchmark (YCSB) [4] is a well-known standard benchmark that tests storage systems with different mixes of read (single-record gets and range scans) and write (single-record inserts and updates) workloads. However, YCSB benchmark reports do not always paint the required accurate

picture on a distributed system's performance and scalability because the amount of stored data and the generated artificial workload is typically not representative for the workload imposed by today's real world applications. Indeed, application architects are faced with the following challenges:

- There is *no one-size-fits-all solution* to select and configure storage technology for supporting data-intensive applications, because (1) different applications may impose workloads (e.g., in terms of mixing read and write operations, in terms of data aggregation and querying) that are better serviced by a particular type of data storage, (2) the suitability of storage technologies for adaptation may vary, and (3) they come with specific configuration options that are not (and cannot be) represented in standard benchmarks.
- The run-time environment and conditions of a specific data-intensive application may not allow for a *statically optimal configuration*, for example because of unpredictable deployment settings in a cloud-based deployment, but also because of the dynamics of the application itself, as *workloads evolve at run-time*. This may jeopardize the effectiveness of performance enhancing tactics (e.g., performance hits of secondary indices for write-heavy workloads).
- Yet dynamic reconfiguration may not pay off: applying a performance tactic at run-time may trigger *a temporary overhead and a delay* before the tactic becomes effective, and the cost of run-time adaptation may not outweigh the benefit if the window of opportunity is short lived.

Selecting the best performing technology and configuring the deployment for scalability and tuning parameters for an optimal service delivery are challenging tasks that should not be re-executed from scratch by knowledgeable application architects. This task should be addressed and facilitated by special purpose middleware.

In this paper, we present a Performance and Scalability Tactics Architecture (PaSTA) and implementation (see Figure 1 for a conceptual view) of a configuration (deployment time) and a dynamic re-configuration (run-time) middleware layer for data-intensive applications that select and map performance and scalability tactics in terms of application specific characteristics and needs by monitoring create, read, update and delete (CRUD) as well as query and aggregation operations. Our middleware design absorbs the know-how through supervised machine learning based on data gathered in performance and scalability experiments in two real-world, industrial application cases that impose complementary Quality of Service (QoS) requirements. The contributions of this paper are threefold:

1. We have investigated and quantified the impact of different performance and scalability tactics, including vertical and horizontal scaling, sharding, caching, replication, in-memory stores and secondary indices. These benchmarking results and newly collected monitoring information is used to automatically tune at runtime the configuration of NoSQL systems through supervised machine learning on streaming monitoring data with Adaptive Hoeffding Trees.
2. The impact analysis sketched above is at the basis of a middleware support layer that offers mapping capabilities to associate these high-level tactics, evolving application workloads, and preferences of the architect with the configuration interfaces of a range of underlying NoSQL technologies, with representative platforms for key-value pair, column-based and document-based stores, as depicted in Table 1.
3. We have validated and applied our solution in the context of industry-level case studies of data-intensive applications where performance and scalability are key. The acquired insights have been embedded into the design of our PaSTA middleware. The evaluation demonstrates that the machine learning based mapping by our middleware can adapt NoSQL systems to unseen workloads of data-intensive applications.

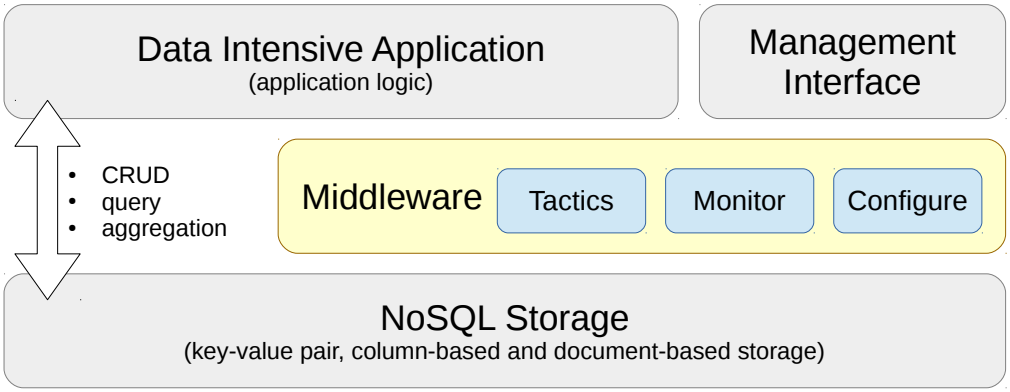

Figure 1. Performance and Scalability Tactics Architecture (PaSTA) middleware with reconfiguration layer.

Ultimately, the role of our PaSTA middleware is to identify and apply NoSQL system configurations for multiple concurrent optimization objectives (e.g., throughput, latency, cost) based on a quantitative assessment of various performance and scalability tactics. Our middleware solves this problem at runtime by monitoring the data growth, changes in the read/write/query mix at run-time, as well as other system metrics that are indicative of sub-optimal performance. Our middleware employs supervised machine learning on historic and current monitoring information and corresponding configurations to select the best combinations of high-level tactics and adapt NoSQL systems to evolving workloads.

Related work on autonomic computing contains point solutions that optimize some of the presented configuration tactics in a more finely grained and sometimes more sophisticated way. The presented middleware in this paper services the comprehensive goal of statically and dynamically managing and tuning the configuration of distributed storage systems that support data-intensive applications that utilize distributed NoSQL technology.

The remainder of the paper is structured as follows. We discuss two real world case studies and the corresponding QoS requirements in Section 2. In Section 3 we review representative NoSQL storage systems and the distributed deployment infrastructures. Section 4 elaborates on the characterization of workloads and the mapping of system configurations. In Section 5, we explain how the acquired insights have been embedded in the design of our middleware and how our prototype configures NoSQL technology based on high-level tactics and preferences imposed by the application architect. We illustrate the feasibility and performance of our solution in Section 6. In Section 7, we discuss and compare with related work. We conclude in Section 8.

Table 1. Performance and scalability tactics in contemporary NoSQL storage systems.

	MongoDB	**Cassandra**	**CouchDB**	**Elasticsearch**	**Riak**	**HBase**	**Redis**
Storage Type	**Doc**	**Col**	**Doc**	**Doc**	**Key-Val**	**Col**	**Key-Val**
Sharding	✓	✓	✓	✓	✓	✓	✓
Replication	✓	✓	✓	✓	✓	✓	✓
Secondary index	✓	✓	✓	✓	✓	✗	✓
In-memory mode	✓	✓	✗	✗	✓	✗	✓
Cache tuning	✓	✓	✗	✓	✗	✓	?
Quorums	✓	✓	✗	✓	✓	✗	✗
Compression	✓	✓	✓	✓	✓	✓	?

2. Motivating Use Cases

In this section, we document two real world industry use cases of data-intensive applications with complementary QoS requirements. These case studies emerge from industrial collaboration and are the motivating examples for this work. We highlight some of the challenging aspects application owners are facing while aiming to deliver predictable performance guarantees and/or a highly responsive customer experience.

2.1. Fraud Analytics in Anywhere-Commerce

The first use case involves an industry offering innovative, fast and secure payment services to its "anywhere-commerce" clients (e.g., brick-and-mortar shop owners and online retail merchants). The payment products handle electronic transactions with credit cards, voucher and gift cards, micro- and mobile payments, and stored value accounts. Their objective is to improve the overall end-user experience and identify fraudulent transactions in real-time by customizing the payment transaction flow based on the context of the customer, vendor, products and transaction value. The risk assessment of such context-dependent payment transactions is subject to hard response time constraints. Decisions should be taken without any noticeable delay for the end-user or merchant. The online fraud analytics is based on fraud rules that merchants can customize in terms of their own business needs:

- **Blacklists/whitelists:** shipping and billing countries, IP address ranges, credit card PAN numbers, countries derived from IP address and/or card number, and so forth, will affect the risk that is dynamically associated with a particular transaction.
- **Velocity rules with sliding windows:** In addition, the number of transactions and the cumulative amount spent per customer, the IP address, card/account number and so forth over given period do constitute a second data source for online risk assessment.
- **Other thresholds:** Additional factors are taken into account, for example the number of credit cards per IP address, and IP addresses per card/account number in absolute values and over a given period.

When violated, each of these fraud rules contributes to a risk score for the transaction that will influence the remainder of the payment flow. The overall risk assessment of a single transaction (excl. network delay) should take less than 10 ms in 99.9% of the transaction requests. Additionally, to account for seasonal variations in the spending behavior (e.g., summer sales), merchants can tune their thresholds and reevaluate historic transactions offline at a rate of at least 10,000 transactions per minute (depending on the Service Level Agreement (SLA)).

Replicating data to an in-memory only storage node eliminates time consuming flushes from memory to disk and compression. This tactic can help in reducing the variability in the request latency to meet the stringent QoS constraints. Additionally, a Time To Live (TTL) secondary index can automatically discard old records to efficiently evaluate the velocity rules. Clearly, these tactics are essential to configure the storage capabilities to support the application level SLA.

2.2. Interactive Situational Awareness

The second application case involves software components that enable the rapid assembly of applications that offer high-performance geospatial situational awareness. The resulting products are used in mission critical systems that handle a large variety and volume (terabytes) of static and moving data covering relevant geographical information. The live data is produced by an ever increasing number of remote devices and sensors, and is stored and processed by high-end enterprise back-end systems. Relevant examples of live data and large-scale historic data sets include car (http://www.ndw.nu/en), air (http://www.flightradar24.com) and marine (http://www.marinetraffic.com) traffic information, or geotagged metadata in maps [5].

A typical scenario is an incident near a harbor (e.g., a hazardous chemical spill) where emergency response teams must quickly identify vessels that are heading towards the incident, or vessels that left the region hours ago. With interactive visualization being a key concern, the objective is to provision the right data with minimal latency to decision makers. Such geospatial queries should be executed in less than a second, and the system must be able to process 10,000 concurrent updates of vehicles or other sensors per second.

Geospatial applications typically retrieve records from data stores holding terabytes of data with queries having location, region or distance constraints. Sharding can distribute the workload for such sophisticated queries. For large data sets, spatial secondary indices are critical to guarantee efficient record retrieval. If not supported, one can construct a geohash [6] based primary key (with a drawback: queries with other fields used as filters require inefficient scanning through all records). Given that the relevant information — and hence the nature of the geospatial queries — highly depends on the role of the stakeholder and the situation at hand, it is clear that dynamic optimization at run-time is key. Again, specific tactics are essential to configure and dynamically re-configure the storage capabilities to support the application needs.

3. NoSQL Deployment Environment Overview

This section elaborates on the 3 NoSQL storage systems that we used as storage backends, and the infrastructures on which these technologies and our middleware were deployed.

3.1. Representative NoSQL Storage Systems

The previous industry cases have been implemented on top of three NoSQL technologies for comparison, and these can be considered to be representative for essential NoSQL categories: Riak for key-value pairs, HBase for column-based technologies and MongoDB for document-based platforms.

3.1.1. Riak

Riak 2.1.4 is a key-value NoSQL database inspired by Amazon's Dynamo architecture [7] that adopts a masterless architecture (see Figure 2) in which every node in the cluster is capable of serving read and write requests. Requests are routed to nodes through standard load balancing. Riak offers replication with consistent hashing and automatically shards data to ensure even distribution (based on mapping each object to a point on the edge of a circle). Depending on the backend used, Riak supports secondary indices through tagging objects, but recommends Solr for indexing instead.

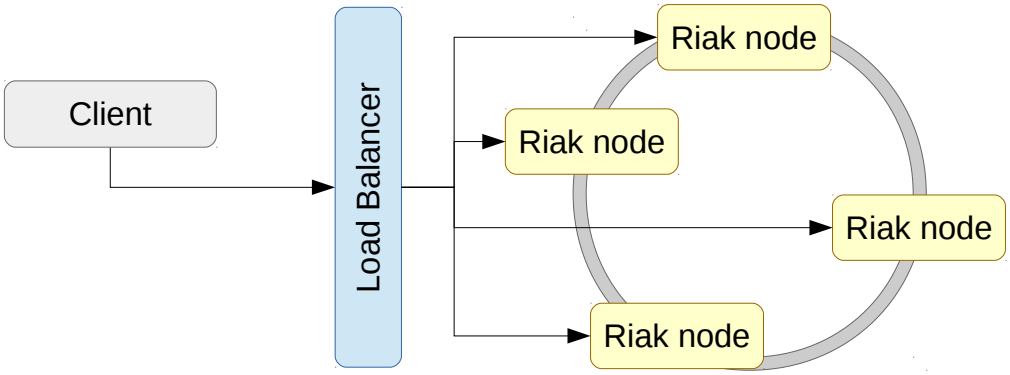

Figure 2. Deployment of a Riak key-value pair cluster.

3.1.2. HBase

Figure 3 depicts our HBase 1.2.0 setup, a distributed column family database providing low-latency random reads and writes on top of the Hadoop Distributed File System (*HDFS*). A clustered deployment has one *HMaster* (master) and multiple *Region Servers* (slaves) that serve data for reads and write. Clients communicate with the *Region Servers* directly. The *BlockCache* read cache stores

frequently read data in memory with a least recently used (LRU) eviction policy. The *MemStore* write cache stores new data which has not yet been written to disk. Each region server hosts multiple regions and stores the data in HDFS in *HFile* format. *ZooKeeper* maintains the server state of the cluster.

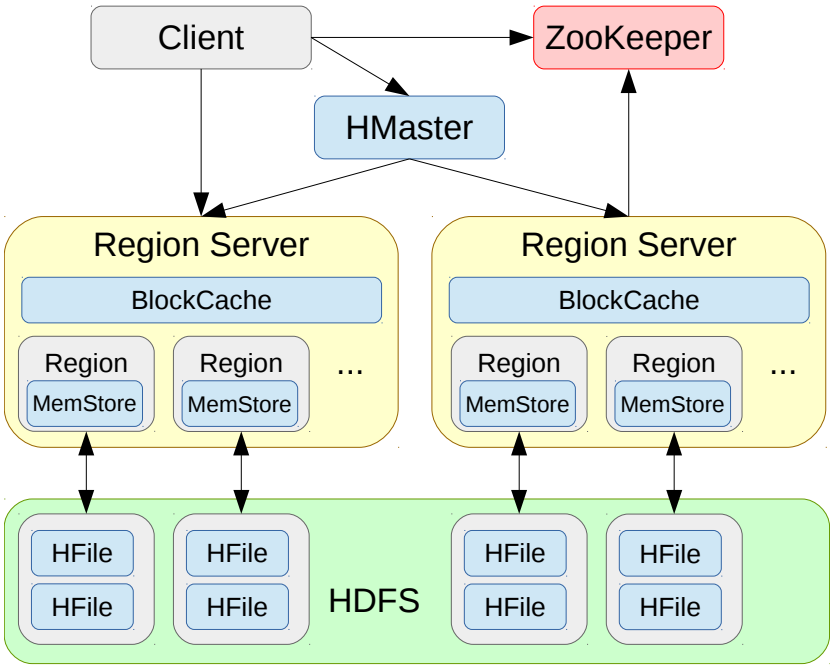

Figure 3. Deployment of an HBase column store cluster.

3.1.3. MongoDB

The last system that has been tested was MongoDB Enterprise 3.4. The deployment setup is shown in Figure 4, both a stand-alone MongoDB instance, and a sharded MongoDB cluster. The *mongod* nodes are the nodes that contain the actual data in the database. In the clustered setup all data is distributed across three *mongod* shard nodes, without replication. The *mongos* node is a proxy that is aware of the data placement. Depending on the query, this node forwards the query and update operations to one or more shards.

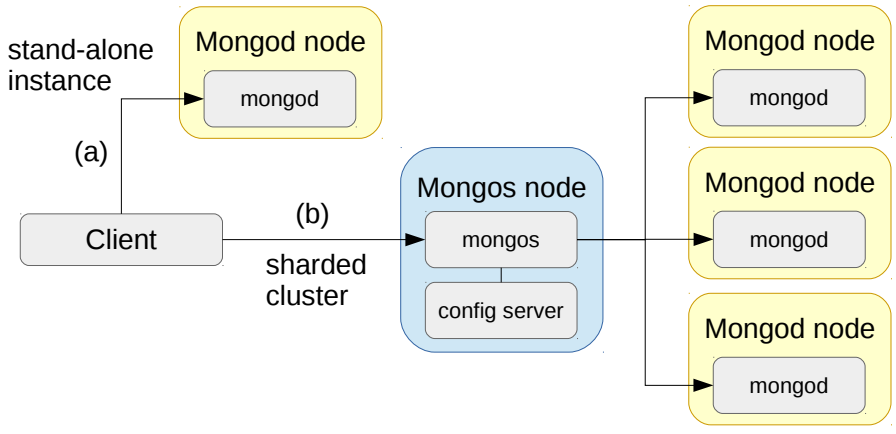

Figure 4. Deployment of (**a**) a single stand-alone MongoDB document store and (**b**) a sharded MongoDB cluster.

3.2. Distributed Deployment Infrastructures

The three NoSQL storage systems and our PasTA middleware are evaluated on three different deployment infrastructures: on high-end Dell PowerEdge servers, on low-end Dell OptiPlex desktop

machines and on low-end virtual machines in a multi-tenant OpenStack private cloud environment with eight compute nodes. An overview of the specification of these machines is shown in Table 2. The cloud environment is optimized for a high degree of resource sharing. One physical CPU can be mapped to multiple virtual CPUs, depending on the workload of the cluster. The desktop machines are performance-wise comparable to the virtual machines in the private cloud. However, native deployments operate with performance isolation and without virtualization overhead or influences of the underlying distributed file system of the private cloud stack.

Table 2. Hardware configuration and specification of the 3 deployment environments.

	Dell PowerEdge R620 (Server)	**Dell OptiPlex 755 (Desktop)**	**OpenStack (vm)**
Deployment	Native	Native	Virtual machine
OS	Ubuntu 16.04 (64-bit)	Ubuntu 16.04 (64-bit)	Ubuntu 16.04 (64-bit)
CPU	2 × Intel Xeon E5-2650 (8 cores) at 2 GHz	Intel Core 2 Duo E6850 (2 cores) at 3 GHz	2 × vCPU
Memory	64 GB	4 GB	4 GB
Hard drive	2 × 900 GB SAS 6 Gbps 10 K RPM	250 GB Seagate Barracuda 7200.10 ATA	virtual
Network	1 Gigabit	1 Gigabit	3 × 1 Gigabit

4. Workload Characterization and System Configuration Mapping

The performance and scalability of the above data-intensive applications not only depend on the distributed NoSQL storage system deployment and configuration, but also on the workload that the data-intensive application imposes on the underlying storage system in terms read/write and query operations.

However, selecting the best performing technology, configuring the deployment for scalability and tuning parameters at runtime for an optimal service delivery remain challenging tasks. To tune distributed NoSQL storage systems for performance and scalability, our PaSTA middleware monitors the data growth, changes in the read/write/query mix at run-time, as well as other system metrics that are indicative of sub-optimal performance and SLA violations. Using supervised machine learning on historic and current monitoring information and corresponding configurations, our middleware then maps application workloads on distributed system configurations to adapt to evolving workloads. The overall approach is depicted in Figure 5.

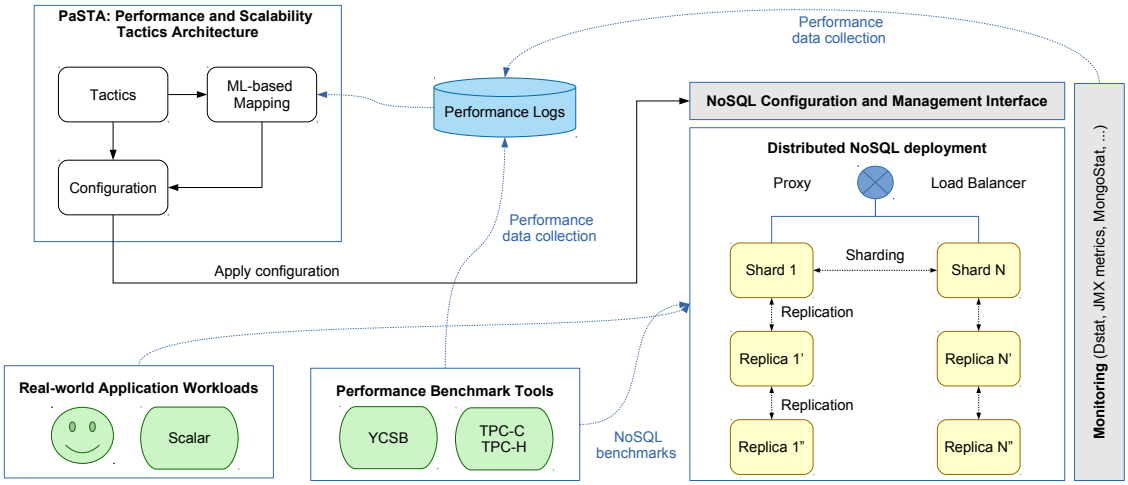

Figure 5. Machine learning based mapping of performance and scalability tactics through runtime configuration on NoSQL systems.

4.1. Data Collection and Feature Extraction

To validate the practical feasibility of our middleware, we will collect system metric statistics of (1) performance benchmarking tools and (2) those of genuine end-users of the application cases to test with realistic workloads and produce convincing experimental evidence under different operating conditions and distributed deployment environments.

4.1.1. Application Agnostic Performance Benchmarks

The motivation to leverage monitoring information from application agnostic benchmarking tools is that they usually cover a broader spectrum of albeit simulated application workloads. This way, our middleware can tap into coarse grained insights to deal with unseen workloads of real world applications. To quantify the effect of the aforementioned trade-offs in practice in an application agnostic manner, the YCSB benchmarking tool (https://github.com/brianfrankcooper/YCSB) has been applied in different experiment setups varying the above tactics. This tool is widely used in research to compare the performance of distributed storage systems, and allows simulating a certain amount of users (client threads) that execute CRUD (create, read, update and delete) operations on a specific storage system under 6 standard and built-in core workloads.

A: 50% read; 50% update; request distribution=zipfian
B: 95% read; 5% update; request distribution=zipfian
C: 100% read; request distribution=zipfian
D: 95% read; 5% insert; request distribution=latest
E: 95% scan (0 to 100 records with uniform probability); 5% insert; request distribution=zipfian
F: 50% read; 50% read-modify-write; request distribution=zipfian

4.1.2. User and Application Specific Workloads

There are several limitations with generic technology benchmarking tools to make well-informed adaptation or reconfiguration decisions. For example, they cannot give any guarantees that the application specific QoS requirements of the industry cases in Section 2 will be met. Furthermore, they usually only test the common denominator of storage system functionality, usually CRUD operations based on the primary key field. There is no way to systematically compare the impact of secondary indices, aggregations or more complex queries. Also, client-side benchmarks do not monitor resource consumption on the servers (CPU, memory, network), making it difficult to explain bottlenecks in terms of NoSQL technologies vs. resource saturation. Tools like YCSB initialize a database with a preset number of records and executes the CRUD operations with a fixed ratio. In practice, the size of the data set and the ratio of read vs. write operations evolve continuously.

That is why our middleware not only uses monitoring information from synthetic workloads, but also traces of real world application specific workloads from genuine end-users. To anticipate a growing user base and increasing amounts of data, we use Scalar [8] to automate the behavior of users and replay realistic application workloads in a controlled manner and monitor the performance and scalability of the distributed storage system in different configurations.

4.1.3. Feature Extraction

Our middleware monitors resource consumption, the data growth, changes in the read/write/query mix at run-time, as well as other system metrics that are indicative of sub-optimal performance with a variety of tools. For example, we use Dstat (http://dag.wiee.rs/home-made/dstat/) for system monitoring, and NoSQL database specific performance monitoring tools (e.g., *mongostat* for MongoDB) to collect run-time statistics and metrics about the transactions itself (e.g., latency and throughput, cache usage, read vs. write mix).

The feature set we use as input to our machine learning pipeline in order to map application workloads on NoSQL system configurations consists of the following features:

- **Resource consumption:** For each node in the cluster, we collect system metrics about CPU, memory and network usage and this aggregated over different sliding windows (5 s, 30 s and 5 min).
- **Data transactions:** Amount of read, scan, write, update and query operations, their duration, size of data processed, again aggregated over the same time intervals.
- **Storage system configuration and transition:** We continuously maintain two sets of attributes (describing the performance and scalability tactics), the first one representing the current configuration, and the second list representing the intended system state. This way we keep track of configuration transitions.
- **Key performance indicators:** These numeric attributes indicate to what extent the application specific SLAs have been met (or violated).
- **Status information:** Amount of users and data in the NoSQL system, and whether the current system configuration is stable or changing (including the ongoing duration of the change in seconds).

With the above set of attributes, we obtain a continuous operational overview of the state and performance of the distributed NoSQL system. For a cluster deployment of 5 nodes, we obtain about 200–230 attributes that are fed to the machine learning algorithm for configuration mapping. The reason for this large amount of attributes is that certain categorical features (e.g., the current and planned storage system configuration) are expanded with one-hot encoding into a format that makes them more suitable for classification and regression algorithms.

4.2. Machine Learning Based Configuration Mapping

In Section 2, we identified several performance and scalability tactics for the given applications cases. However, these configuration changes are only effective under specific operating conditions. For example, replication for write-intensive workloads increases the network traffic between the replica nodes jeopardizing the application throughput, and it may also negatively influence the query latency due to quorum or consistency management and conflict resolution. Sharding also incurs a performance overhead as the proxy lookup service has to determine to which shard an operation must be directed or even merge query results from different shards. Secondary indices are mostly beneficial for read-intensive workloads (imposing minimal index maintenance overhead) when the indices themselves can be stored in memory. As such, there is a trade-off between using memory for (partial) caching or secondary indices. Furthermore, changing workloads may render existing secondary indices useless. We therefore use a machine learning approach, as the manual mapping of (re)configurations in uncertain operating conditions is not straightforward.

As the monitoring data and extracted features have a streaming nature, we use Adaptive Hoeffding Trees (AHT) [9] as an incremental classifier that is capable of learning from massive data streams. Traditional decision tree methods like C4.5 require that all training data is available in memory so that the data can be processed in multiple passes. Also, many methods do not handle well data distributions that change over time. These observations have motivated our decision for adopting the AHT classifier, implemented in the MOA machine learning library [10], for the decision logic in our PaSTA middleware. It only does one pass over the collected information, with low processing times per record, low memory consumption, and not requiring any disk accesses. Furthermore, compared to the original Hoeffding Trees (a.k.a. VFDT, an abbreviation of *Very Fast induction of Decision Trees*) and enhanced Concept-adapting VFDTs [11] that can adaptively learn from data streams that drift over time, AHTs have the benefit that they do not need any user-defined configuration parameters to guess how fast a stream will change. Also, they have been shown to perform better than CVFDTs. These properties make them ideal candidates to effectively map NoSQL configurations for data-intensive application workloads that evolve over time.

5. The PaSTA Middleware Implementation

The PaSTA middleware in Figure 6 includes three layers; At the top level, the middleware offers a management dashboard to the application owner, supporting (a) the observation of performance and QoS/SLA compliance that is achieved, and (b) potentially the approval of configuration changes that are proposed by the core of the middleware. The *Performance Monitoring* component of our middleware collects resource consumption metrics for the nodes in the cluster using off-the-shelf performance monitoring tools. Additionally, NoSQL database specific performance monitoring tools (e.g., *mongostat* for MongoDB) collect run-time statistics and metrics about the transactions itself (e.g., latency and throughput, cache usage, read vs. write mix). The *Unified Runtime Metrics* collects, aggregates, normalizes and interprets metrics, and signals events regarding saturated resources (i.e., above a 90% capacity threshold for cpu, memory, network and hard drive). The normalized metrics are computed with Esper (http://www.espertech.com/), a Complex Event Processing (CEP) engine. The raw measurements and normalized metrics are fed into the *SLA Policy Monitoring* component which verifies compliance with QoS/SLA goals configured by the application architect in the *SLA Policy Administration* component through the *Management Dashboard*. The *SLA Policy Monitoring* component autonomously triggers the *Configuration Selection and Mapping* component that leverages knowledge and input from the *Tactics Knowledge Base* and the *Configuration Repository* of current and previous configurations.

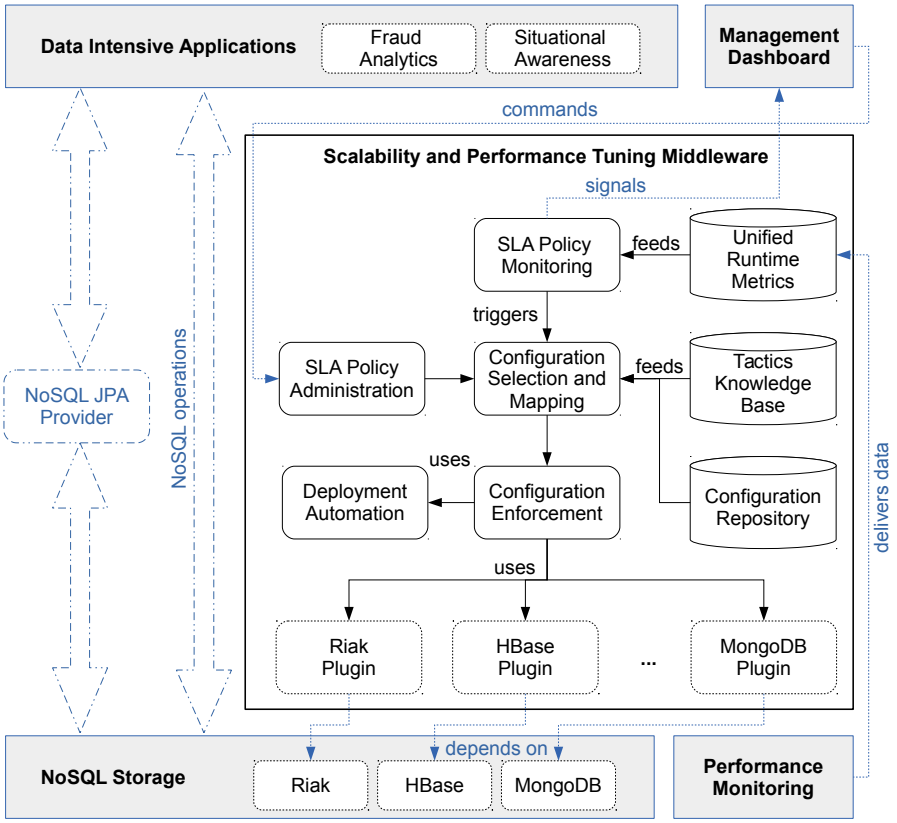

Figure 6. PaSTA: middleware to tune performance and scalability.

At the bottom level, the middleware includes monitoring support to feed the core functionality with online measurements to actually quantify the performance and scalability characteristics of the running applications. This essential monitoring support is not tightly integrated with the core of the middleware, in order to flexibly leverage upon existing and emerging third part monitoring components.

At the heart of the middleware architecture is a core that can, in principle, autonomously configure and reconfigure the storage layer to optimize performance characteristics of the data-intensive application at run-time. The *Tactic Knowledge Base* component maintains generic knowledge about the benchmarks and a list of tactics that have a positive or negative impact on the *throughput* in terms of NoSQL operations per second, the *latency distribution* of these NoSQL operations, resource consumption and type of operation. Last but not least, the *Deployment Automation* component provisions and configures replication or shard nodes in the NoSQL cluster using the Chef [12] deployment and configuration management tool.

The remainder of this section will elaborate in more detail on the *Configuration Selection and Mapping* component that embeds the machine learning logic, and more specifically the Adaptive Hoeffding Trees algorithms for streaming data classification, to map application workloads and operational conditions on NoSQL system configurations.

5.1. Bootstrapping the Adaptive Hoeffding Tree Machine Learning Model with Application Agnostic Workloads

Adaptive Hoeffding Trees need sample data to learn which configurations are more effective. As this parameter-free classifier can automatically deal with sliding windows and concept drifts in data streams, we initialize the classifier by means of artificial simulated workloads generated by the YCSB [4] benchmark tool for different distributed NoSQL system configurations. Below follows a subset of the series of benchmark experiments we carried to help bootstrap the adaptive Hoeffding tree model with application agnostic workloads.

Table 3 shows a performance comparison of a stand-alone database system for different technologies, namely MongoDB, HBase and Riak. A surprising observation is that MongoDB performs a lot better in terms of throughput and average latency compared to HBase and Riak. One would assume that these more advanced features of MongoDB would lead in principle to a slower database system, but this is not the case. In fact, Riak performs the worst while it has the simplest data model, that is, a key-value store.

Table 3. Performance of a stand-alone MongoDB, HBase and Riak database on a OptiPlex desktop and PowerEdge server (workload C).

	OptiPlex Desktop		
	MongoDB	**HBase**	**Riak**
Overall throughput	35,198 ops/s	3325 ops/s	2505 ops/s
Average Latency	445 µs	4747 µs	3174 µs
Min Latency	118 µs	211 µs	480 µs
Max Latency	103,615 µs	914,431 µs	164,223 µs
95th Percentile Latency	837 µs	8631 µs	4359 µs
99th Percentile Latency	1859 µs	73,407 µs	26,271 µs
	PowerEdge Server		
	MongoDB	**HBase**	**Riak**
Overall throughput	39,478 ops/s	19,233 ops/s	12,466 ops/s
Average Latency	390 µs	772 µs	1253 µs
Min Latency	236 µs	330 µs	783 µs
Max Latency	127,679 µs	255,103 µs	173,439 µs
95th Percentile Latency	638 µs	1041 µs	2171 µs
99th Percentile Latency	1032 µs	1699 µs	3039 µs

Another interesting observation is that the performance of a database system does not necessarily increase when the database server is deployed on a more powerful machine.

- When moving a stand-alone MongoDB server from an OptiPlex desktop machine (throughput = 35,198 ops/s) to a PowerEdge server (throughput = 39,478), the performance does barely increase. The same holds for the average latency.
- For HBase and Riak, on the other hand, performance increases a lot when moving to the PowerEdge server. HBase, for example, achieves 3325 operations per second on an OptiPlex machine, while it achieves a throughput of 19,233 operations per second on a PowerEdge server. This observation indicates that MongoDB is probably reaching a network boundary, which prevents it from further increasing its throughput.

Figure 7 shows the throughput of a MongoDB database for a given deployment configuration on OptiPlex desktop machines. The different deployment configurations covered in this figure are: a stand-alone configuration, a configuration with three replicas each containing the full dataset and a sharded setup with three nodes where all data is split across these three nodes. The figure shows that the stand-alone setup is able to process a significantly larger amount of requests per second, compared to the distributed setups. Intuitively, the sharded setup should increase the throughput, since all query and modify operations are loadbalanced over the three shards. The results show that this is not true. Even worse, the sharded setup is the poorest performing setup for workload E, while the sharded setup should allow the scan operations to run in parallel on the nodes of the different shards. A second observation is that the replicated setup and the sharded setup perform more or less the same. As a conclusion, sharding the database does not have a performance advantage when all data can fit into the cache of a stand-alone database.

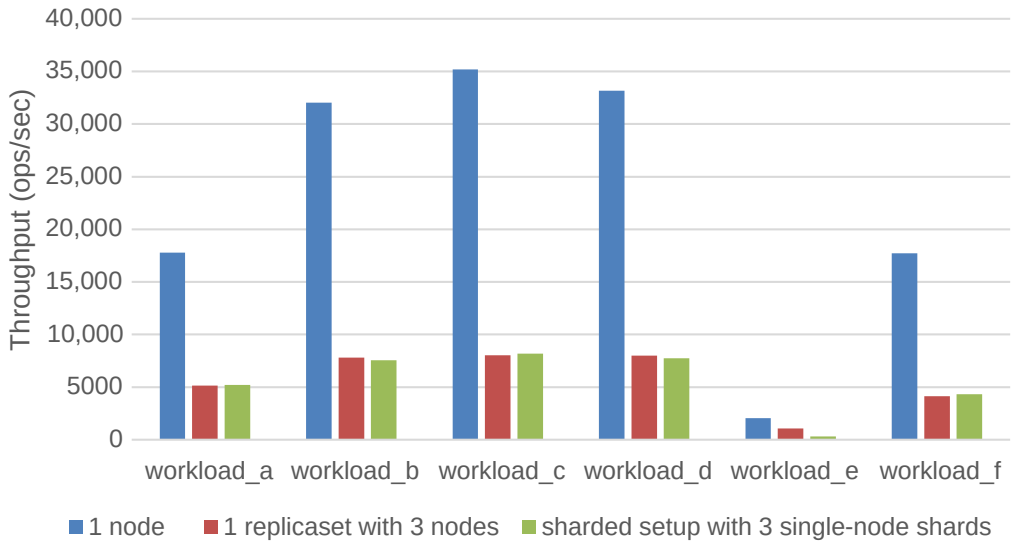

Figure 7. Impact of the deployment configuration on the throughput when the data fits in the cache of a stand-alone MongoDB database.

Figure 8 shows the results for the same experiment in case the data does no longer fit into the cache of a stand-alone MongoDB database system, but it does fit into the cache when the data is sharded across three nodes. The three shards increase the total cache size of the database system by a factor of three. The workloads C, D, E and F show an increased throughput compared to the previous experiment. Of course, these workloads represent the more read-oriented workloads, which benefit from the larger cache size. This indicates that the performance of a data store not only depends on the workload, but obviously also on the amount of data present in the store.

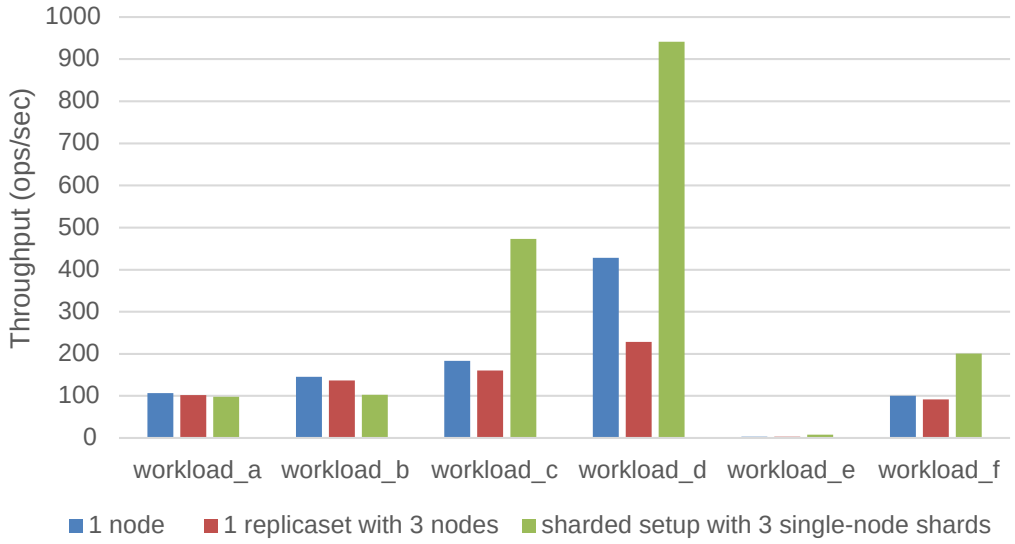

Figure 8. Impact of the deployment configuration on the throughput when the data does not fit in the cache of a stand-alone MongoDB database.

Figure 9 shows the impact of the workload on the latency of read and update operations that the workload consists of. The figure gives an overview of the average latency, the 95th and the 99th percentile of the latency for workloads A, B and C. These workloads consist of different distributions of read and update operations. The results show that the 99th percentile of the latency of workload A is significantly larger than the 99th percentile latency of workloads B and C. This indicates that the variance on the latency increases when the proportion of modify operations of the workload increases. This can be explained by the fact that these operations require more disk I/O, while a lot of read operations can be served from the database cache.

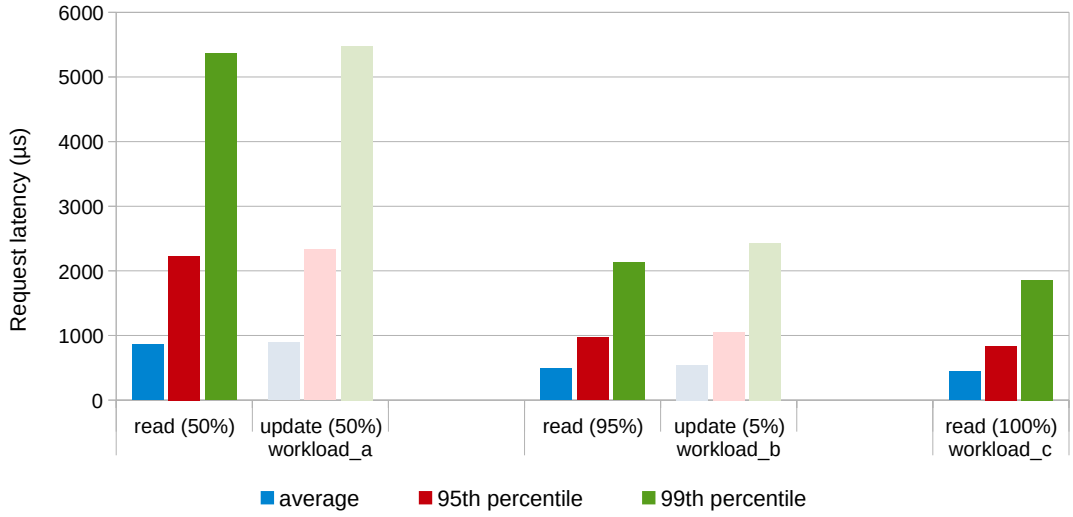

Figure 9. Impact of read/update mix on operation latency in a stand-alone MongoDB server deployed on an OptiPlex machine.

5.2. Dynamic Reconfiguration for Real World Application-Specific Workloads

The above YCSB benchmarks are simulations of workloads and therefore cannot accurately represent the performance and scalability impact of the motivating case studies of Section 2. However, they do provide indications of what the relative influence is of one performance enhancing tactic compared to another, and how they interfere with one another. The insights and observations were elicited from the previous off-line NoSQL technology benchmark experiments:

- Memory and caching are key, that is, a faster machine does not necessarily increase the throughput.
- A less complex data model does not necessarily lead to better performance.
- Sharding is not useful if the data fits in the cache of a stand-alone system.
- A virtualized environment leads to more latency jitter for write operations that require disk access.

While these observations might be trivial or obvious, the goal is to use the measurements of the above benchmark experiments as baseline to pre-train the adaptive Hoeffding decision trees. As our supervised machine learning method can handle classification of data streams with concept drifts, our middleware will then let the real world application workloads automatically further optimize the adaptive decision tree model. The advantage of this approach is two-fold. Not only can the mapping and reconfiguration decision logic within our middleware automatically adapt from artificial workloads to real world application workloads, but it can achieve the same capability at runtime when the workload of the data-intensive application itself evolves over time. The dynamic modifications that are currently supported include:

- **Secondary index:** Add a secondary index for frequently read attributes. The index computation takes time. The index will consume additional memory and will slow down write operations to maintain the secondary index.
- **Secondary index:** Remove unused secondary indices. The impact is mainly noticeable for write-heavy workloads, but to some extent also for read operations that can benefit from the recovered memory that is now available for caching.
- **Sharding:** Add a shard to share to keep more data in memory. Spawning a new virtual machine takes 10 s, and reshuffling the data to the new shard has an impact on the internal network of the cluster.
- **Replication:** Add a node to a replica set to distribute the processing workload over more compute nodes. This tactic is mainly beneficial for read-heavy workloads with little overhead to guarantee data consistency across nodes.
- **In-memory storage engine:** Increase the request throughput by replicating the data to a node with more processing power or more memory for caching the data and secondary indices. Replicate to an in-memory node to reduce the latency variability.

Some of the tactics are difficult to achieve with certain technologies. For example, HBase does not support secondary indices within the same table. That is why the *Tactics Knowledge Base* groups supported tactics per technology, with customized implementations of each tactic offered by a NoSQL platform specific plugin.

6. Evaluation with Application-Specific Workloads

Based on the two application cases defined earlier, we will demonstrate the feasibility of our middleware to dynamically customize the configuration of NoSQL systems at run-time to meet QoS constraints. We first initiate genuine application-specific workloads for a growing number of users with the Scalar [8] benchmark framework in order to gain insights when the maximum capacity of a system is reached. With Scalar, we are able to create realistic application workloads by taking real-life user interactions with the system, and replaying those on a much large scale for many more users and/or transactions to test the limits of a particular system configuration. Scalar is built around Gunther's Universal Scalability Law [13] (USL) that combines (a) the initial linear scalability of a system under increasing load, (b) the cost of sharing resources, (c) the diminishing returns due to contention, and (d) the negative returns from incoherency into a model that defines the relative capacity $C(N)$:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}, \quad (1)$$

where N represents the scalability of the system in terms of the number of concurrent requests, α represents the contention penalty, and β defines the coherency penalty, with $0 \leq \alpha, \beta < 1$. To benchmark the scalability, N is incremented on a fixed configuration.

As such, the main advantage of Scalar over YCSB, is that the former can create realistic application workloads in controlled conditions that will stress test the underlying NoSQL storage system and trigger application specific SLAs violations.

6.1. Fraud Analytics

The first case deals with effective fraud analysis with stringent time constraints. Table 4 illustrates the difference in latency of MongoDB storage engine technology measured with Scalar benchmarking at about 40,000 primary key-based read requests per second (the maximum capacity of the system according to the USL) over a Gigabit connection to the NoSQL datastore. While these results show better results for the In-Memory engine, the overall results are very conservative as the actual fraud analytics application runs on the same host as the MongoDB instance, avoiding network traffic between the application and the NoSQL store.

Table 4. Performance comparison on a PowerEdge server of 2 MongoDB stand-alone storage engines.

Percentile Latency	WiredTiger	InMemory
50th Percentile	547 µs	472 µs
90th Percentile	877 µs	614 µs
95th Percentile	1082 µs	711 µs
99th Percentile	1693 µs	1002 µs
99.9th Percentile	3273 µs	1778 µs
99.99th Percentile	42,071 µs	38,932 µs

The black- and whitelists are loaded once (per merchant) at startup and do not require any further NoSQL operations. However, the velocity rules require *count()* and *sum()* computations over transactions within a given time window for several attributes in the request (e.g., the number of transactions and the cumulative amount for an IP address). Figures 10 and 11 show the capacity of that system (in terms of concurrent users or threads each submitting 10 requests per second) when there is a secondary index (Figure 11) or not (Figure 10) on that attribute. Notice the orders of magnitude of scalability difference in terms of concurrent requests (20 vs. 2500 concurrent users). However, more important are the results on the end-to-end latency distribution as shown in Table 5 for about 200 transactions per second. Note that these numbers also include the network delay due to end-to-end measurements at the client.

Table 5. Impact secondary index on latency.

Latency	No Index	Secondary Index
50.0th Percentile	2667 ms	4.778 ms
99.9th Percentile	26,539 ms	13.642 ms

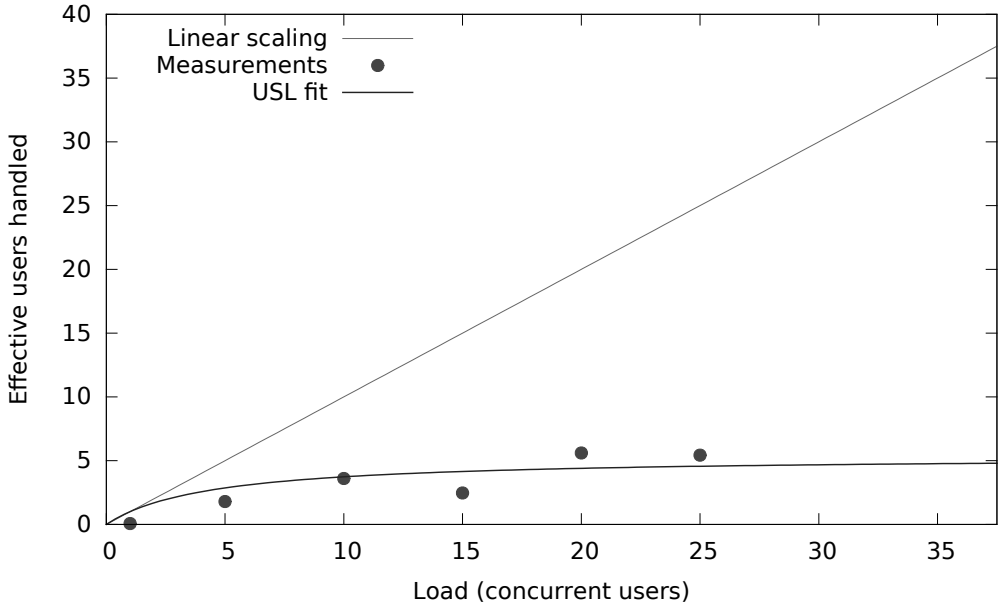

Figure 10. Fraud velocity rule without secondary index.

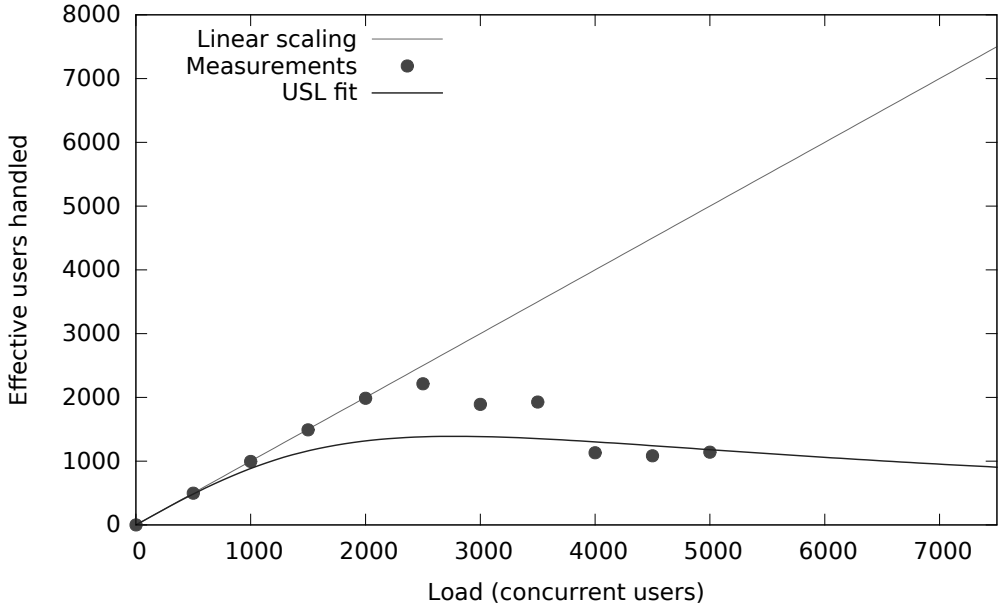

Figure 11. Fraud velocity rule with secondary index.

We increased the workload by adding more concurrent realistic user (or merchant) workloads to the fraud analytics application to investigate the decision and impact of PaSTA's reconfiguration automation. We add a new fraud rule relying on an unindexed attribute when the system is under a load of about of 4 million records/min. The performance counters are reset, and the first evaluation of the new rule will violate the SLA policy. Figure 12 depicts what happens when the SLA of 10 ms per transaction is violated. The adaptive Hoeffding tree classifies the current state of the NoSQL configuration, and decides that adding a secondary index is the best tactic to improve the query performance to evaluate the fraud velocity rules. It takes about 9 s before the secondary index becomes effective for those records matching the sliding windows of the velocity rules.

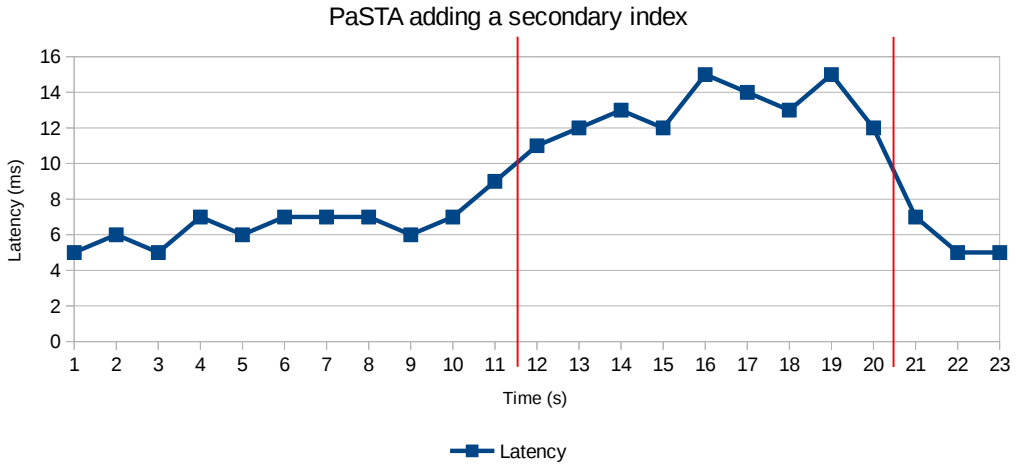

Figure 12. Time required before secondary index becomes effective.

Increasing the workload further with more concurrent transactions, and with all secondary indices set, the single node can no longer fulfill the SLA constraint of 99.9% in less than 10 ms (excluding network delay) beyond 1100 requests per second. The Hoeffding decision tree suggests to scale out with sharding (based on merchant field in the record). The partial migration of half of the data to the new shard takes about 23 s, so the cost of this tactic is higher than the secondary index. Another reason for PaSTA (in fact the underlying Hoeffding tree) to make this decision is the slightly worse latency improvement due to the overhead of routing incoming requests to different shards.

6.2. Situational Awareness

The second use case with geospatial data investigates the effect of changing the read/write workload at run-time, and the performance effect of query operations when the amount of data in the increases. Due to the nature of geospatial queries, MongoDB WiredTiger was used as it was the only one offering the necessary distance computation functionality and query APIs out-of-the-box. The experiment is based on flight radar data with an average insert rate of spatially indexed 10000 flight location updates per second as required according to the SLA, the feasibility of which is demonstrated in Figure 13.

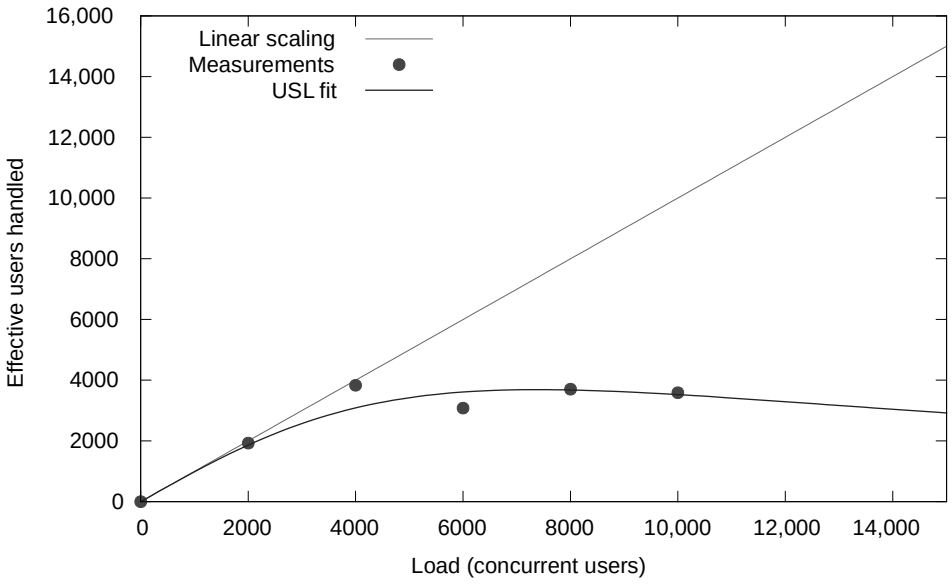

Figure 13. Linear scalability for location updates up to 4000 concurrent users (threads at 10 updates/s).

At a sustained rate of 10,000 updates per second, Figure 14 shows the results of geospatial queries of flight records near a random airport, with results capped to 100 entries sorted per distance and filtered with a maximum distance of 100 km. The maximum capacity is reached at 200 concurrent users (or 2000 requests per second). With MongoDB's default cache size of 60% of system RAM minus 1 GB, that is, 37 GB for our PowerEdge server, Table 6 and especially the 90% percentile show the effect of spatial queries hitting the hard disk even if for 70 GB of data still more than half of the queries can be handled in less than a second (per SLA).

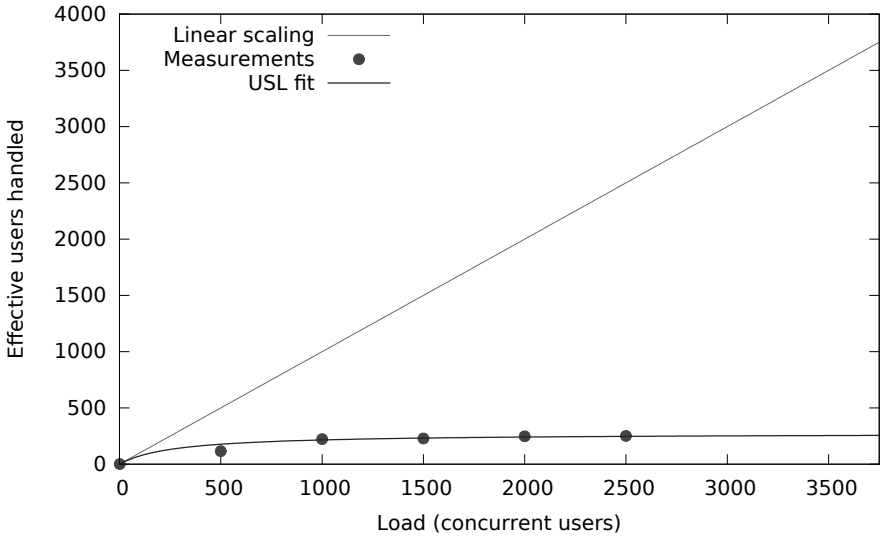

Figure 14. Maximum capacity reached for 200 concurrent users (threads at 10 queries/s).

Table 6. Impact of data size on spatial query latency.

Latency	35 GB Data	70 GB Data
50.0th Percentile	55.5 ms	432 ms
90.0th Percentile	303 ms	9932 ms

The decision tree within PaSTA evaluated sharding to be the most effective tactic to reduce request latency. Sharding to a second PowerEdge server based on the flight identifier (MongoDB does not support sharding based on a spatial index) reduced the number of cache misses. Figure 15 shows that re-allocation of chunks takes several minutes to complete (due to the amount of geospatial data to be transferred).

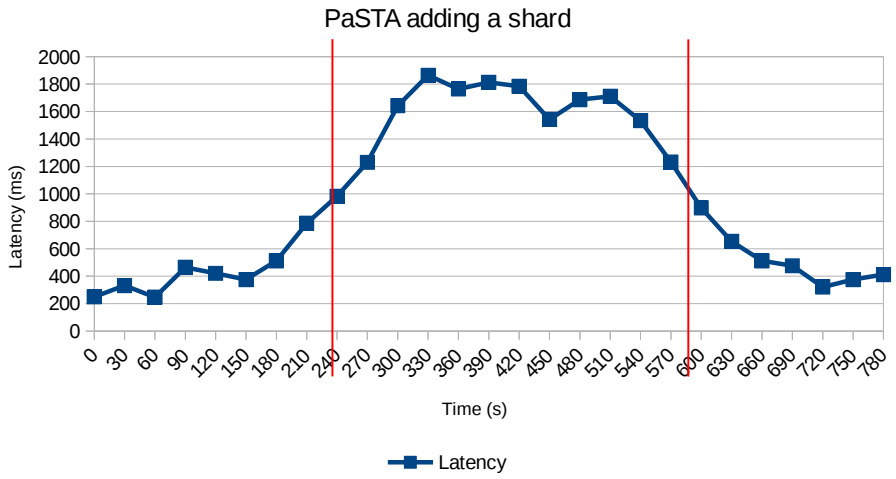

Figure 15. Time required before sharding becomes effective.

6.3. Performance Impact of Adaptive Hoeffding Trees

One of the reasons for using Adaptive Hoeffding Trees to map application workloads to system configurations is that it processes performance metrics only once and does not need to maintain all metrics and other data attributes in memory for additional processing later on.

The memory overhead of AHT for the first application case is shown in Figure 16, indeed illustrating that it remains relatively stable over time. The only scenario where memory does increase is when new nodes are added to the cluster, as it creates new system metrics to be processed by the PaSTA middleware.

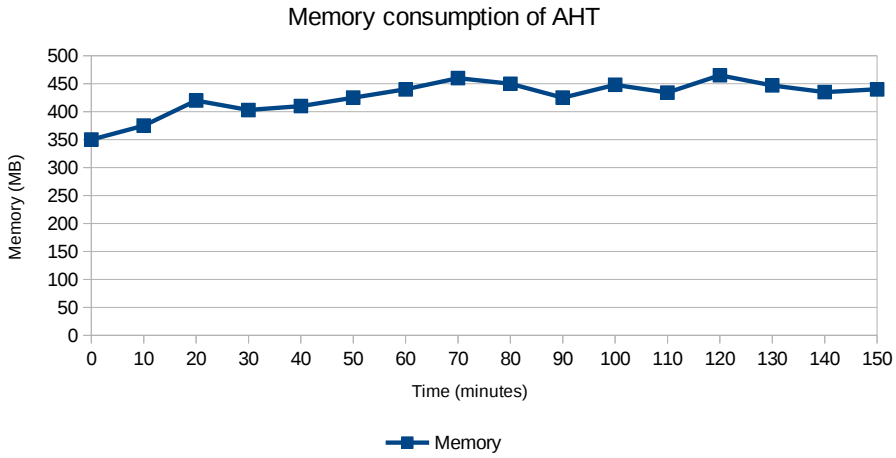

Figure 16. Time required before sharding becomes effective.

We also evaluated the computational overhead of AHT, but found that the impact was relative small (less than 10 ms for a single dynamic reconfiguration decision), and lower than the overhead imposed by the middleware components that transform all incoming metrics to a uniform representation for feature extraction and SLA policy monitoring.

6.4. Adaptation to Evolving Workloads

We evaluated the capability to adapt to evolving NoSQL workloads in two different ways. Although the Adaptive Hoeffding Trees were initialized with synthetic workloads rather than real world users, we found the PaSTA middleware to be quite capable of making adequate reconfiguration decisions on real world application workloads, although we did not carry out an exhaustive search to explore whether the decision taken was always the most optimal.

A clarification for why the AHT was able to adapt to the new workload, is because the decision of which tactic to initiate depends on the *relative* comparison with other tactics regarding their costs and benefits. If, however, the relative ranking would be different between synthetic workloads and real world application workloads or if the data-intensive application would be deployed on a completely different distributed infrastructure, it is likely that PaSTA will make a subpar reconfiguration decision. However, as the cost and impact of new reconfigurations is also continuously monitored, these values will be fed into the AHT to improve its adaptation decisions in the future.

Another scenario that we cannot rule out, but for which we found no evidence in our experimentation, is the possibility of inefficient oscillations due to PaSTA going back and forth between two different NoSQL configurations. An example is the creation of secondary indices that (a) create an performance overhead for write operations and (b) reduce memory availability for caching. If these are unused for read operations, PaSTA will delete them, but trigger their re-creation whenever a particular query could possibly benefit from it (as in the velocity rules for the fraud analytics application case). However, it is fairly straightforward to circumvent this problem by adding an additional cost to trigger such unwanted oscillations.

7. Related Work

The middleware that is presented in this work essentially has three capabilities. First, it can be used to observe and monitor the performance of NoSQL technologies under a broad variety of circumstances and workloads. Secondly, it can be used to statically configure the data storage facility based on well-known tactics. Third, it can be used in its full strength by applying the monitoring capabilities at production time, thus enabling dynamic selection of appropriate tactics that are mapped and configured at run-time. We discuss related work from these three perspectives, and refer to related work applied in the NoSQL space, as well as other data storage platforms such as RDBMS. This section is therefore structured as follows. Section 7.3 presents related work regarding dynamic optimizations of storage systems. Section 7.1 gives an overview of middleware that allows static configuration of data storage systems.

7.1. Static Workload-Based Optimization

Khattab et al. [14] present a framework to tune a database system layer-by-layer, that is, both at the software level as well as at the hardware level. They argue that tuning the different configuration parameters involved, is a complex task and requires a lot of expertise. In this work, a neural network is trained to capture the complex relationship between the input parameters (DBMS parameters, performance parameters, workload vectors and server utilization) and the output parameters, which is the predicted number of transactions the database system will be able to execute.

NoSQL system are also used to execute complex aggregation operations on the stored data using mapreduce techniques. Gencer et al. [15] say that different hardware, system and software configuration, can have a significant impact on the time it takes to perform a mapreduce computation. Since the search space of the amount of possible configuration is very large, trying all different combinations on a real system is unfeasible, especially if hardware reconfiguration are required. They apply a combination of coarse-grained filtering, that prunes away the configurations that will definitely not be able to achieve the required performance, followed by a finer-grained filtering based on a simulation of the actual computation with the proposed configuration parameters. Herodotou et al. [16] did similar work in optimizing the performance of mapreduce jobs. They present an approach named *just-in time job optimization* which determines proper configuration parameter by applying rules of thumb and dynamic profiling of a mapreduce job. The middleware in this paper also monitors the workload based on the type and ratio of read/write operations involved.

7.2. Autonomic Computing

Autonomic computing is a decade old paradigm that focusses on distributed computing systems that can manage themselves [17,18], with self-optimization being one of the key self-* properties of autonomic systems. Utility functions are commonly applied for this purpose, both for the initial deployment of an application and its dynamic reconfiguration. Tesauro et al. [19] explored utility functions as a way to enable a collection of autonomic elements to continually optimize the use of computational resources in a dynamic, heterogeneous environment. Later work by Deb et al. [20] investigated how utility functions can be used to achieve self-optimized deployment of computationally intensive scientific and engineering applications in highly dynamic and large-scale distributed computing environments. Utility functions have also found their way into the cloud computing space [21,22] where they are used to manage virtualized computational and storage resources that can scale on demand. The problem with utility functions is that their definitions require a fair amount of domain-specific knowledge to be effective, especially in the presence of many performance and scalability trade-offs as in our application cases.

Esteves et al. [23] investigated the use of fuzzy nets to specifically deal with real-time processing demands for continuous and data-intensive processing in complex data-intensive workflows. Similar to our work, their solution for workload aware resource management for NoSQL systems like HBase

also relies on machine learning techniques. They present Fluxy to deliver controlled and augmented performance, rationalization of resources, and task prioritization, albeit relying on a different machine learning based decision logic. Van Aken et al. [24] also investigated the use of both supervised an unsupervised machine learning to automatically tune database management systems. Compared to our work, their solution called OtterTune focusses on MySQL and Postgres DBMS systems, and can recommend more fine grained configuration optimizations in less than 60 min. For our application cases, the recommendation and enforcement should be completed in a matter of minutes.

7.3. Dynamic Workload-Based Optimization

This section presents related work on different middleware platforms that allow dynamic reconfiguration of NoSQL and RDBMS systems. Section 7.3.1 addresses autonomous addition and removal of both replicas and sharding in a distributed database system based on the needs implied by the actual run-time workload. Section 7.3.2 presents related work on dynamically tuning trade-off between consistency and performance in eventually consistency storage systems. Finally, Section 7.3.3 presents related work on autonomous index creation.

7.3.1. Sharding and Replication

Sharding and replication are one of the primary tool by which NoSQL systems achieve horizontal scalability and high availability. Choosing the data placement wisely can substantially contribute to increased performance, as shown by Curz et al. [25]. Their paper specifically focuses on the data placement in the HBase system. They show that, for example, collocating the read intensive data records and collocating the write intensive data records can increase performance significantly as these shards can be optimized for their specific workload characteristics. Curino et al. [26] target the performance problems that occur with executing distributed transactions in a sharded or replicated database system. Their approach tries to analyse the workload of the user to collocate records that are often used together in a distributed transaction, preventing the need for an expensive distributed transaction the next time. Our work also applied workload-aware reconfiguration similar to the work in References [25,26]. Our approach is relatively more coarsely grained as we focus on full workloads (e.g., by starting from the relative contribution of write-operations.) We expect that dealing with individual records or groups of records cannot be addressed by general purpose reconfiguration middleware - though this theme definitely demands for additional research.

7.3.2. Eventual Consistency

Achieving strong consistency in a distributed database system is not easy, since it involves a performance overhead and includes risk for data unavailability in the presence of node failure. For this reason, many NoSQL systems support eventual consistency and allow tuning the required level of consistency. Quorum-based systems allow the user to define the amount of node that have to confirm the read and write operation before reporting success to the client. The most common consistency levels are one, quorum and all. For these level respectively one, a majority or all the replicas have to confirm the read or write operation. If the amount of nodes in the read consistency level plus the amount of nodes in the write consistency level are larger or equal than the replication factor, strong consistency can be achieved. Otherwise, only eventual consistency is guaranteed. Couceiro et al. [27] focus on achieving strong consistency, with a near-optimal quorum configuration for the current workload. The hotspot records are assigned a customized quorum configuration. The remainder of the records is assigned a collective quorum configuration, to keep the configuration management scalable. So, for read-intensive records, a lower read consistency level is assigned to the record and a higher write consistency level and the other way around. McKenzie et al. [28] argue that the current choices for the consistency levels is too discrete, to allow the consistency-based SLAs. They propose a system that probabilistically chooses between read and write consistency level one and quorum, depending on the defined SLA. The level of consistency

is a configuration feature that is set in the management dashboard, and mapped onto underlying technology. This middleware currently does not offer any tactics to automatically tune the read/write consistency level or quorum.

7.3.3. Index Creation

The need for autonomously tuning database systems is not something specific to NoSQL based systems. In related work effort has been spent on more dynamic configuration management. SMIX, is an autonomous indexing infrastructure developed by Volgt et al. [29] to support dynamic index creation as well as the destruction of indexes. SMIX also supports partial index creation in case only a certain part of a column requires indexing. The authors argue that the more agile application development and the increasing amount of data make workload changes more frequent. As a consequence, it would be unfeasible to define the indexes manually every time significant workload changes occur. The middleware proposed in this paper monitors attribute usage in queries to decide upon creating or deleting secondary indices, but cannot automatically configure special purpose ones like TTL, partial or compound secondary indices.

7.4. Threats to Validity

Within the frame of this research, several choices were made with respect to not only the NoSQL technologies that were tested, but also the machine learning technique used to autonomously tune the database platforms. In this section, we will offer some critical reflections on these choices and the implications for our work.

First of all, PaSTA uses Adaptive Hoeffding Trees, as implemented by the MOA software framework, to map the best configuration for a given NoSQL workload. The MOA framework offers other algorithms, such as the meta-classifier "Bagging using Adaptive-Size Hoeffding Trees", and there may be others that would fit PaSTA's objectives. A more systematic analysis of different machine learning models and methods could lead to a better tuning and optimization strategy. In fact, various AutoML frameworks, such as Auto-WEKA [30] and Auto-sklearn [31], have been proposed to help automate the best model selection and hyperparameter tuning of the corresponding machine learning models. However, the underlying Bayesian optimization based exploration can be a time consuming process, and it is likely that the best model selection and configuration may in the end depend on the typical workload of the use case or that it is hard to generalize which machine learning methods work best overall.

Secondly, in terms of the NoSQL types we considered in this work, we only evaluated key-value pair, column-oriented and document-based databases, as depicted in Table 1. We did not examine graph data stores, such as Neo4j, Titan, OrientDB or JanusGraph. The main reason is that none of our use cases demanded the complexity of a graph data store. Furthermore, some of these graph data stores − such as JanusGraph − support various storage backends such as Cassandra or HBase. Similar to the YCSB tool that is not well-equipped to test the capabilities of graph databases as it favors portability and simplicity over relevance, PaSTA cannot yet fully exploit all the configuration capabilities either. The configuration space to optimize the performance or the scalability of distributed graph data stores is far more challenging due to the structure of the underlying data, and well-known tactics such as sharding, are not always available.

Third, zooming in on the NoSQL types we did consider, there are obviously many representative candidates that could have been tested instead, possibly with different outcomes. For example, Redis and DynamoDB are popular key-value stores, whereas Cassandra is another widely used column-oriented store. The availability of NoSQL technologies is growing by the day. The feasibility to support alternative candidates, as well as the assessment of newer versions or different flavors of the tested NoSQL technologies is an opportunity for future work. However, it should be pointed out that to support additional NoSQL databases, PaSTA needs to have an understanding of the configuration options to enhance the performance and scalability of the database, and how these

options may interfere with one another. Additionally, to be effective, the framework also needs to be able to monitor the queries to make well-informed reconfiguration decisions at runtime in response to evolving workloads. Perhaps contrary to other benchmarking tools such as YCSB that aim for portability, extending our PaSTA framework with new functionality requires more effort than merely implementing some abstract interfaces.

Last but not least, PaSTA relies on application agnostic benchmarking workloads generated by YCSB and one may question this dependency. YCSB benchmarks lead to valuable insights that help quantify the relative impact of one configuration over another. For example, increasing the replication factor will in general slow down database requests. Indeed, for NoSQL systems that support quorum, i.e., $R + W > N$ with N the replication factor, at least R replicas must be accessed by a read operation and at least W replicas must be accessed by a write operation. Whenever N increases, so must R and/or W. Application agnostic benchmarks with YCSB can help characterize the impact for a given workload mix of read and write operations and a concrete distributed deployment environment. In practice, it would be possible for PaSTA to not leverage YCSB's assessments, but this may lead to ineffective reconfiguration decisions when limited evidence is available to predict the impact of a reconfiguration tactic.

7.5. Summary

In summary of this section, we argue that the presented middleware in this paper services the comprehensive goal of statically and dynamically managing and tuning the configuration of the storage systems that support data-intensive applications that utilize NoSQL technology. Clearly, related work contains point solutions that optimize some of the presented configuration tactics in a more finely grained and sometimes more sophisticated way. It is our goal to gradually improve the presented middleware architecture and its implementation to absorb promising improvements, while remaining manageable for deployers of data-intensive applications. The latter topic is an additional and important avenue for future research.

8. Conclusions

This paper has presented a middleware architecture and implementation that supports data-intensive applications in selecting and mapping performance tactics for the configuration of a wide variety of NoSQL technologies. The middleware thus leverages upon application specific characteristics to deliver optimal performance. Our architecture has been driven by two industry relevant case studies and embeds knowledge generated in extended benchmarking.

These benchmarking results and streaming monitoring information are used to bootstrap and automatically tune at runtime the configuration of NoSQL systems through supervised machine learning using Adaptive Hoeffding Trees. This particular stream mining and classification algorithm is able of associating high-level reconfiguration tactics, evolving application workloads, and capabilities of underlying NoSQL technologies. The evaluation with realistic application workloads demonstrates that Adaptive Hoeffding Trees can adapt NoSQL systems to unseen workloads of data-intensive applications, while maintaining a low and stable memory and computational overhead on the system. PaSTA enables static and dynamic configuration of multiple NoSQL technologies by dealing with an extensive set of performance tactics.

The evaluation of the presented middleware, that is covering and addressing a lot of complexity, is of course non-trivial. We argue that its application and the benchmarking with real world industry cases delivers a convincing illustration of the middleware's potential. Further experimentation and usage reports are beyond the scope of a single paper and subject of our next steps and future work. Other avenues for further research include improved support for other NoSQL data stores and NoSQL types. By supporting multiple candidates for a given NoSQL type within the same CAP theorem [32] classification, a more extensive evaluation could lead to new insights whether optimization tactics can be reused by means of transfer learning [33], hereby eliminating the need for YCSB benchmarking.

The outcome may be that other machine learning methods beyond Adaptive Hoeffding Trees are better suited for this purpose. A systematic analysis and comparison of other classification techniques is needed to support this capability to generalize across NoSQL technologies.

Author Contributions: Conceptualization, methodology, software, and validation, D.P.; writing—original draft preparation, D.P.; writing—review and editing, D.P. and W.J.; supervision, D.P. and W.J.; project administration, D.P.; funding acquisition, W.J. Both authors have read and agreed to the published version of the manuscript.

Funding: This research is partially funded by the Research Fund KU Leuven.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Klein, J.; Gorton, I.; Ernst, N.; Donohoe, P.; Pham, K.; Matser, C. Performance Evaluation of NoSQL Databases: A Case Study. In Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems, Austin, TX, USA, 1 February 2015; pp. 5–10. [CrossRef]
2. Mozaffari, M.; Nazemi, E.; Eftekhari-Moghadam, A.M. Feedback control loop design for workload change detection in self-tuning NoSQL wide column stores. *Expert Syst. Appl.* **2020**, *142*, 112973. [CrossRef]
3. Hillenbrand, A.; Störl, U.; Levchenko, M.; Nabiyev, S.; Klettke, M. Towards Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), Dallas, TX, USA, 20–24 April 2020; pp. 133–138.
4. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154. [CrossRef]
5. Haklay, M.M.; Weber, P. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Comput.* **2008**, *7*, 12–18. [CrossRef]
6. Fox, A.; Eichelberger, C.; Hughes, J.; Lyon, S. Spatio-temporal indexing in non-relational distributed databases. In Proceedings of the 2013 IEEE International Conference on Big Data, Santa Clara, CA, USA, 6–9 October 2013; pp. 291–299. [CrossRef]
7. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, Stevenson, WA, USA, 14–17 October 2007; pp. 205–220. [CrossRef]
8. Preuveneers, D.; Heyman, T.; Berbers, Y.; Joosen, W. Systematic scalability assessment for feature oriented multi-tenant services. *J. Syst. Softw.* **2016**, *116*, 162–176. [CrossRef]
9. Bifet, A.; Gavaldà, R. Adaptive Learning from Evolving Data Streams. In Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII, Lyon, France, 31 August–2 September 2009; pp. 249–260. [CrossRef]
10. Bifet, A.; Holmes, G.; Kirkby, R.; Pfahringer, B. MOA: Massive Online Analysis. *J. Mach. Learn. Res.* **2010**, *11*, 1601–1604.
11. Hulten, G.; Spencer, L.; Domingos, P. Mining time-changing data streams. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 26–29 August 2001; pp. 97–106.
12. Marschall, M. *Chef Infrastructure Automation Cookbook*; Packt Publishing: Birmingham, UK, 2013.
13. Gunther, N.J. *Guerrilla Capacity Planning—A Tactical Approach to Planning for Highly Scalable Applications and Services*; Springer: Cham, Switzerland, 2007. [CrossRef]
14. Khattab, A.; Algergawy, A.; Sarhan, A. MAG: A performance evaluation framework for database systems. *Knowl.-Based Syst.* **2015**, *85*, 245–255. [CrossRef]
15. Gencer, A.E.; Bindel, D.; Sirer, E.G.; van Renesse, R. Configuring Distributed Computations Using Response Surfaces. In Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, 7–11 December 2015; pp. 235–246. [CrossRef]
16. Herodotou, H.; Lim, H.; Luo, G.; Borisov, N.; Dong, L.; Cetin, F.B.; Babu, S. Starfish: A Self-tuning System for Big Data Analytics. CIDR. Available online: https://www-db.cs.wisc.edu/cidr/cidr2011/Papers/CIDR11_Paper36.pdf (accessed on 3 August 2020).

17. Koehler, J.; Koehler, J.; Giblin, C.; Giblin, C.; Gantenbein, D.; Gantenbein, D.; Hauser, R.; Hauser, R. On Autonomic Computing Architectures. In *Research Report (Computer Science) RZ 3487(#99302)*; IBM Research: Zurich, Switzerland, 2003.
18. Huebscher, M.C.; McCann, J.A. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.* **2008**, *40*, 1–28. [CrossRef]
19. Tesauro, G.; Kephart, J.O. Utility Functions in Autonomic Systems. In Proceedings of the First International Conference on Autonomic Computing, Washington, DC, USA, 17–18 May 2004; pp. 70–77.
20. Deb, D.; Fuad, M.M.; Oudshoorn, M.J. Achieving self-managed deployment in a distributed environment. *J. Comp. Methods Sci. Eng.* **2011**, *11*, 115–125. [CrossRef]
21. Hu, Y.; Wong, J.; Iszlai, G.; Litoiu, M. Resource provisioning for cloud computing. In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, Toronto, ON, Canada, 2–5 November 2009; pp. 101–111. [CrossRef]
22. Koehler, M.; Benkner, S. Design of an Adaptive Framework for Utility-Based Optimization of Scientific Applications in the Cloud. In Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, Chicago, IL, USA, 5–8 November 2012; pp. 303–308. [CrossRef]
23. Esteves, S.; Silva, J.a.N.; Carvalho, J.a.P.; Veiga, L. Incremental Dataflow Execution, Resource Efficiency and Probabilistic Guarantees with Fuzzy Boolean Nets. *J. Parallel Distrib. Comput.* **2015**, *79*, 52–66. [CrossRef]
24. Van Aken, D.; Pavlo, A.; Gordon, G.J.; Zhang, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 14–19 May 2017; pp. 1009–1024. [CrossRef]
25. Cruz, F.; Maia, F.; Matos, M.; Oliveira, R.; Paulo, J.; Pereira, J.; Vilaça, R. Met: workload aware elasticity for nosql. In Proceedings of the 8th ACM European Conference on Computer Systems, Prague, Czech Republic, 15 April 2013; pp. 183–196.
26. Curino, C.; Jones, E.; Zhang, Y.; Madden, S. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* **2010**, *3*, 48–57. [CrossRef]
27. Couceiro, M.; Chandrasekara, G.; Bravo, M.; Hiltunen, M.; Romano, P.; Rodrigues, L. Q-OPT: Self-tuning Quorum System for Strongly Consistent Software Defined Storage. In Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, 7–11 December 2015; pp. 88–99.
28. McKenzie, M.; Fan, H.; Golab, W. Continuous Partial Quorums for Consistency-Latency Tuning in Distributed NoSQL Storage Systems. *arXiv* **2015**, arXiv:1507.03162.
29. Voigt, H.; Kissinger, T.; Lehner, W. Smix: Self-managing indexes for dynamic workloads. In Proceedings of the 25th International Conference on Scientific and Statistical Database Management, Baltimore, MD, USA, 29–31 July 2013; p. 24.
30. Kotthoff, L.; Thornton, C.; Hoos, H.H.; Hutter, F.; Leyton-Brown, K. Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA. In *Automated Machine Learning: Methods, Systems, Challenges*; Hutter, F., Kotthoff, L., Vanschoren, J., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 81–95.
31. Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.T.; Blum, M.; Hutter, F. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning: Methods, Systems, Challenges*; Hutter, F., Kotthoff, L., Vanschoren, J., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 113–134.
32. Brewer, E. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* **2012**, *45*, 23–29. [CrossRef]
33. Jamshidi, P.; Velez, M.; Kästner, C.; Siegmund, N.; Kawthekar, P. Transfer Learning for Improving Model Predictions in Highly Configurable Software. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*; IEEE Press: Piscataway, NJ, USA, 2017; pp. 31–41.