Quality Attribute-Guided Evaluation of NoSQL Databases: A Case Study

John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe
Architecture Practices, Software Solutions Division
Carnegie Mellon University Software Engineering Institute
Pittsburgh, PA, USA
{jklein, igorton, nernst, pd}@sei.cmu.edu

Kim Pham, Chrisjan Matser
Telemedicine and Advanced Technology Research Center
US Army Medical Research and Material Command
Frederick, MD, USA
kim.solutionsit@gmail.com, cmatser@codespinnerinc.com

***Abstract*— For software developers, the selection of a particular NoSQL technology imposes a specific distributed software architecture and data model, making the technology selection difficult to defer. NoSQL database technologies provide high levels of performance, scalability, and availability by simplifying data models and supporting horizontal scaling and data replication. Each NoSQL product embodies a particular set of consistency, availability, and partition tolerance (CAP) tradeoffs, along with a data model that reduces the conceptual mismatch between data access and data storage models. This means technology selection must be done early, often with limited information about specific application requirements, and the decision must balance speed with precision, as the NoSQL solution space is large and evolving rapidly. In this paper we present the method and results of a study to compare the architecturally-relevant characteristics of three NoSQL databases for use in a large, distributed healthcare organization. We reflect on some of the fundamental difficulties of performing detailed technical evaluations of NoSQL databases specifically, and big data systems in general, that have become apparent during our study.**

Keywords—NoSQL, distributed databases, technology evaluation

I. Introduction

The exponential growth of data in the last decade has fueled a new specialization for software technology, namely that of *big data*, software systems [1]. At the heart of big data systems are a collection of database technologies that are more simple and lightweight, and provide higher scalability and availability than traditional relational databases [2]. Pioneering efforts from Internet-born organizations such as Google and Amazon [3][4], along with those of numerous other big data innovators, have created a variety of open source and commercial database technologies for organizations to construct and operate massively scalable, highly available data repositories.

These highly scalable "NoSQL" databases [5] are typically designed to scale horizontally across clusters of low cost, moderate performance servers. They achieve high performance, elastic storage capacity, and availability by replicating and partitioning data sets across a cluster. Each database specifies its own proprietary data model and query language, as well as specific mechanisms for achieving distributed data consistency and availability. Prominent examples of NoSQL databases include Cassandra, Riak, and MongoDB.

Due to the inherent diversity in NoSQL technologies, database selection must be carefully considered. When a particular database and its data model is chosen for a application, the associated consistency and distribution models imposed by the database have a pervasive impact on the design of the associated applications [6]. Hence, the selection of a particular NoSQL database must be made early in the design process and is difficult and expensive to change downstream. In other words, NoSQL database selection becomes a critical architectural decision for big data systems.

COTS product selection has been extensively studied in software engineering [7][8][9]. In complex technology landscapes with multiple competing products, organizations must balance the cost and speed of the technology selection process against the fidelity of the decision [10]. While there is rarely a single 'right' answer in selecting a complex component for an application, selection of inappropriate components can be costly, reduce downstream productivity due to rework, and even lead to project cancelation. This is especially true for large scale, big data systems due to their complexity and the magnitude of the investment.

In this context, COTS selection of NoSQL databases for big data applications presents several unique challenges:

- This is an early architecture decision that must be made with inevitably incomplete definitions of requirements;
- The capabilities and features of NoSQL products vary widely, making generalized comparisons difficult;
- Prototyping at production scale is usually impractical, as this would require hundreds of servers, multi-terabyte data sets, and thousands or millions of clients;
- The solution space is changing rapidly, with new products constantly emerging, and existing products releasing several versions per year with ever-evolving feature sets.

We faced these challenges during a recent project for a healthcare provider seeking to adopt NoSQL technology for an Electronic Health Record (EHR) system. The system supports healthcare delivery for over nine million patients in more than 100 facilities across the globe. Data currently grows at over one terabyte per month, and all data must be retained for 99 years.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE
16 JAN 2015

2. REPORT TYPE
N/A

3. DATES COVERED

4. TITLE AND SUBTITLE
Quality Attribute-Guided Evaluation of NoSQL Databases: A Case Study

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Matser /John Klein Ian Gorton Neil Ernst Patrick Donohoe Kim Pham Chrisjan

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT
unclassified

b. ABSTRACT
unclassified

c. THIS PAGE
unclassified

17. LIMITATION OF ABSTRACT
SAR

18. NUMBER OF PAGES
10

19a. NAME OF RESPONSIBLE PERSON

In this paper, we outline a technology evaluation and selection method we have devised for big data systems.

We then describe a quantitative and qualitative study we performed for the health-care provider described above. We introduce the study context, our evaluation approach, and the results of both extensive performance and scalability testing and a detailed feature comparison. We also reflect on our experience, describing some of the essential and accidental challenges we overcame in conducting this evaluation. The specific contributions of the paper are as follows:

- A rigorous method that organizations can follow to evaluate the performance and scalability of NoSQL databases.
- Performance and scalability results that empirically demonstrate significant variability in the capabilities of the databases we tested to support the requirements of our healthcare customer.
- Practical insights and recommendations that organizations can follow to help streamline a NoSQL database evaluation for their own applications.

II. Related Work

Rigorous evaluation methods support data-driven analysis and insightful comparisons of the capabilities of candidate components for an application. Prototyping as part of component evaluation provides important benefits that include both quantitative assessment of performance and qualitative understanding of other factors related to adoption. Gorton describes a rigorous evaluation method for middleware platforms, which can be viewed as a precursor for our work [10].

Benchmarking of databases is generally based on the execution of a specific workload against a specific data set, such as the Wisconsin benchmark for general SQL processing [11] or the TPC-B benchmark for transaction processing [12]. These publically available workload definitions have long enabled vendors and others to publish measurements, which consumers can then attempt to map to their target workload, configuration, and infrastructure for product comparison and selection. These benchmarks were developed for relational data models, and are not relevant for NoSQL systems.

YCSB [13] has recently emerged as the default data set and workload generator for executing simple benchmarks for NoSQL systems. YCSB++ [14] extends YCSB with more sophisticated, multi-phase workload definitions and support for multiple coordinated clients to increase the load on the database server. There is an emerging collection of published measurements using YCSB and YCSB++, from product vendors [15][16] and from researchers [17][18]. In this project, we built on the YCSB framework, incorporating a more complex data set and workload definition that was specific to our healthcare system requirements.

All of the work discussed above has used "generic" data sets and workloads. In contrast, the T-Check method [19] performs evaluation by defining selection criteria and then prototyping and measuring the candidate technology in the context of use. The results reported by Dede and colleagues [20] is an example of the type of hybrid qualitative and quantitative analysis that we performed, using system-specific data sets and workloads to answer specific questions about the candidate technologies within a particular context of use.

III. Evaluation Method

Our method is inspired by earlier work on middleware evaluation [22][23] and customized to address the characteristics of big data systems. The basic main steps are depicted in Fig. 1 and outlined below:

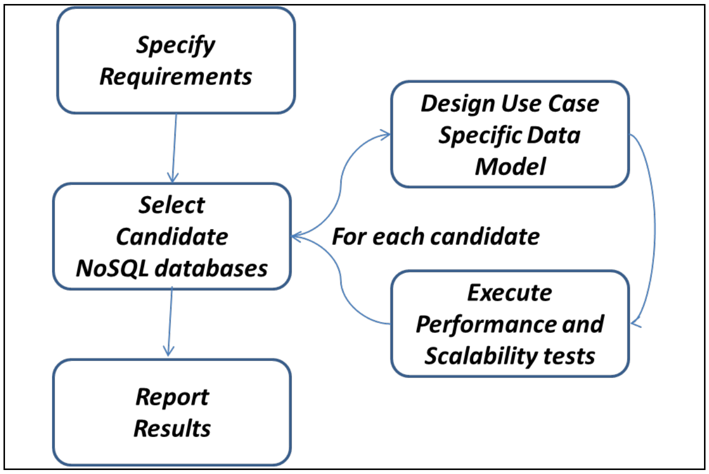

Specify Requirements
Design Use Case Specific Data Model
Select Candidate NoSQL databases
For each candidate
Execute Performance and Scalability tests
Report Results

Fig. 1 Lightweight Evaluation and Prototyping for Big Data (LEAP4BD)

1. **Specify Requirements**: First, we elicit the high priority functional and quality requirements for the system. For big data applications, we focus on specific requirements for performance, scalability, availability, consistency, and security. We also define a use case that is representative of the customer's application domain. The use case defines both a data model and workload that will be used as the basis of performance and scalability assessment in Step 4.
2. **Select Candidate NoSQL Databases**: We select 2-4 candidate NoSQL databases for deeper evaluations. Selection criteria are both contextual (e.g. experience with a specific technology) and technical, and require evaluation of database features against the requirements .
3. **Design Use Case Specific Data Model**: Based on the use case defined for evaluation, we map the application logical data model to the physical model supported by the candidate NoSQL databases. We also deploy the database and load the test data (synthetic or actual) into the database instances.
4. **Execute Performance and Scalability Tests**: We implement a test case driver that executes the specified workload on each database. Load is scaled by increasing the number of concurrent client requests to assess how each database reacts to increased workloads.
5. **Report Results**: The evaluation report includes both qualitative and quantitative results. This details the performance and scalability results we obtained from testing each NoSQL database in a consistent environment.

It also describes how easily the logical data model maps to the specific NoSQL data models that were tested, and the specific features of each database that will influence the identified quality requirements for the application.

In the rest of this paper we describe our experiences applying this method to the EHR project and reflect on the key issues that we faced during the project.

IV. EHR Case Study

A. Project Context

Our customer was a large healthcare provider developing a new electronic healthcare record (EHR) system. This will replace their existing system, which utilizes thick client applications hosted at sites around the world, all connected to a centralized relational database. NoSQL technologies were considered attractive candidates for two specific uses, namely:

- the primary data store for the EHR system
- a local cache at each site to improve request latency and availability

As the customer was familiar with RDMS technology for these use cases, but had no experience using NoSQL, they directed us to focus the technology evaluation only on NoSQL technology.

B. Specifying Requirements

We began the engagement with a stakeholder workshop, using a tailored version of the Quality Attribute Workshop Method [21], to elicit key functional and quality attribute requirements to guide our technology evaluation. These requirements fell in to two broad categories, one quantitative, one qualitative, as follows:

Performance/Scalability: The main quantitative requirements were the ability to easily distribute and replicate data across geographically distributed databases, and to achieve high availability and low latencies under load in distributed database deployments. Hence understanding the inherent performance and scalability that is achievable with each candidate NoSQL database was an essential part of the evaluation.

Data Model Mapping Complexity: Health care systems have clearly defined logical data models that need to be supported by a NoSQL database. As data models vary considerably between NoSQL technologies, an important qualitative requirement was to understand how the health care data model could be supported in the NoSQL alternatives. This required us to evaluate the specific features of each database for data modeling and querying, along with the mechanisms for maintaining consistency in a distributed deployment.

We also worked with the customer to define two driving use cases for the EHR system. These provided the basis for data model evaluation and performance and scalability assessment we performed in subsequent steps in the project. The first use case was retrieving recent medical test results for a particular patient, which is a core EHR function used to populate the user interface whenever a clinician selects a new patient. The second use case was achieving strong consistency for all readers when a new medical test result is written for a patient, because all clinicians using the EHR to make patient care decisions should see the same information about that patient, whether they are at the same site as the patient, or providing telemedicine support from another location.

C. Select Candidate NoSQL Databases

Our customer was specifically interested in evaluating how different NoSQL data models (key-value, column, document, graph) would support their application domain. For this reason we selected one NoSQL database from each category to investigate in detail. We subsequently ruled out graph databases as they did not support the horizontal partitioning required for the customer's requirements. After a short feature assessment of various databases, we settled on Riak, Cassandra and MongoDB as our three candidate technologies, as these are the market leaders in each NoSQL category. In addition to satisfying functional requirements for the EHR application, each of these products is stable and mature, with enterprise technical support available, and hence is compatible with the customer's operational and business constraints.

D. Design and Execute Performance Tests

A thorough evaluation and comparison of complex database platforms requires prototyping with each to reveal the performance and scalability capabilities [10]. To this end, we developed and performed a systematic procedure that provides a foundation for an "apples to apples" comparison of the 3 databases we evaluated. Based on the use cases defined during the requirements step, we:

- Defined a consistent test environment for evaluating each database, including server platform, test client platform, and network topology.
- Mapped the patient record logical model to each database's data model and loaded the resulting database with a large collection of synthetic test data.
- Created a load test client that implements the database read and write operations defined for each use case. This client is capable of issuing many simultaneous requests so that we can analyze how each technology responds as the request load increases.
- Defined and executed test scripts that exerted a specified load on the database using the test client.

We executed each test case on several distributed configurations to measure performance and scalability. These test scenarios ranged from baseline testing on a single server to 9 server instances that sharded and replicated data.

Based on this approach, we are able to produce a consistent set of test results that assess the likely performance and scalability of each database for this customer's EHR system.

V. Prototype and Evaluation Setup

A. Test Environment

The three databases we tested were:

1. MongoDB version 2.2, a document store (http://docs.mongodb.org/v2.2/);

2. Cassandra version 2.0, a column store (http://www.datastax.com/documentation/cassandra/2.0);
3. Riak version 1.4, a key-value store (http://docs.basho.com/riak/1.4.10/).

Prototyping and evaluation were performed on two database server configurations: Single node server, and a nine-node configuration that was representative of a production deployment. A single node test allowed us to validate our base test environment for each database. The nine-node configuration used a topology that represented a geographically distributed deployment across three data centers. The data set was partitioned (i.e. "sharded") across three nodes, and replicated to two additional groups of three nodes each. We used MongoDB's primary/secondary feature, and Cassandra's data center aware distribution feature. Riak did not support this "3x3" data distribution, so we used a configuration where the data was sharded across all nine nodes, with three replicas of each shard stored across the nine nodes.

All testing was performed using the Amazon EC2 cloud (http://aws.amazon.com/ec2/). Database servers were run on "m1.large" instances, with the database data and log files stored on separate EBS volumes attached to each server instance. The EBS volumes were "standard", not provisioned with the EC2 IOPS feature, to minimize the tuning parameters used in each test configuration. Server instances ran the CentOS operating system (http://www.centos.org). The test client was also run on an "m1.large" instance, and also used the CentOS operating system. All instances were in the same EC2 availability zone (i.e. the same cloud data center).

B. Mapping the data model

We used a subset of the HL7 Fast Healthcare Interoperability Resources (FHIR) data model (http://www.hl7.org/implement/standards/fhir/) for our analysis and prototyping. The logical data model consisted of FHIR Patient Resources (e.g., demographic information such as names, addresses, and telephone numbers), and laboratory test results represented as FHIR Observation Resources (e.g., test type, result quantity, and result units). There was a one-to-many relation from each patient to the associated test results. Although this was a relatively simple model, the internal complexity of the FHIR Patient Resource, with multiple addresses and phone numbers, along with the one-to-many relation from patient to observations, required a number of data modeling design decisions and tradeoffs in the data mapping.

A synthetic data set was used for testing. This data set contained one million patient records, and 10 million lab result records. The number of lab results for a patient ranged from zero to 20, with an average of seven. The Patient and Observation Resources were both mapped into the data model for each of the databases we tested.

C. Create load test client

Our test client was based on the YCSB framework [13], which provides capabilities to manage test execution and test measurement. For test execution, YCSB has default data models, data sets, and workloads. We therefore modified YCSB and replaced these default behaviors with implementations specific to our use case data and requests.

We were able to leverage YCSB's built-in capabilities for specifying the total number of operations to be performed and the percentage of read and write operations in the workload. The test execution capabilities also allow the use of multiple execution threads to create concurrent client sessions.

In the measurement framework, for each operation performed, YCSB measures the *operation latency*, which is the time from when the request is sent to the database until the response is received back from the database. The YCSB reporting framework records latency measurements separately for read and write operations. Latency distribution is a key scalability measure for big data systems [4][24], so we recorded both average and 95th percentile values.

We extended the YCSB reporting framework to report *Overall Throughput*, in operations per second. This measurement was calculated by dividing the total number of operations performed (read plus write) by the workload execution time. The execution time was measured from the start of the first operation to the completion of the last operation in the workload execution, and did not include initial setup and final cleanup times.

D. Define and execute test scripts

The stakeholder workshop identified that the typical workload for the EHR system was 80% read and 20% write operations. For this operation mix, we defined a read operation to retrieve the five most recent observations for a single patient, and a write operation to insert a single new observation record for a single existing patient.

Our customer was also interested in using the NoSQL technology as a local cache, so we defined a write-only workload that was representative of a daily load of a local cache from a centralized primary data store with records for patients with scheduled appointments for that day. Finally, we defined a read-only workload that was representative of flushing the cache to the centralized primary data store.

For each database configuration tested, every workload was run three times in order to minimize the impact of any transient events in the cloud infrastructure. For each of these three runs, the workload execution was repeated for a defined range of test client threads (1, 2, 5, 10, 25, 50, 100, 200, 500, and 1000), which created a corresponding number of concurrent database connections. During the testing, the standard deviation of the throughput for any three-run set for a particular thread count never exceeded 2% of the average. We post-processed the results to combine the measurements by averaging across the three runs for each thread count.

Operating with a large number of concurrent database client sessions is not typical for a NoSQL database. Clients usually connect first to a web server tier and/or an application server tier, which aggregates the client operations on the database using a pool of perhaps 16-64 concurrent sessions. However, our prototyping was in support of the modernization of a system that used thick clients with direct database connections, and so our customer wanted to understand the implications of retaining this thick client architecture.

VI. Performance and Scalability Results

We report here on our results for a nine-node configuration that reflected a typical production configuration. As noted above, we also performed testing on a number of configurations for each database under evaluation, ranging from a single server up to a nine-node cluster. The single-node configuration's availability and scalability limitations make it impractical for production use, and so we do not present performance comparisons across databases for this configuration. However, in the following discussion, we compare the single node configuration for a specific database to distributed configurations. This provides insights into the efficiency of that database's distributed coordination mechanisms and guides scalability tradeoffs between adding more nodes versus using faster nodes with more storage.

Defining a test configuration required several design decisions. The first was how to distribute client connections across the server nodes. MongoDB uses a centralized router node, and all clients connected to the single router node. Cassandra's data center aware distribution feature created three sub-clusters of three nodes each, and client connections were spread uniformly across the three nodes in one of the sub-clusters. In the case of Riak, the product architecture only allowed client connections to be spread uniformly across the full set of nine nodes. An alternative might have been to test Riak on three nodes with no replication, however other constraints in the Riak architecture resulted in extremely poor performance in this configuration, and so the nine-node configuration was used.

A second design decision was how to achieve strong consistency, which requires defining both write operation settings and read operation settings [6]. Each of the three databases offered slightly different options, and we explored two approaches, discussed in the next two sections. The first reports results using strong consistency, and the second reports results using eventual consistency.

A. Performance Evaluation – Strong Consistency

The selected options are summarized in TABLE I. For MongoDB, the effect is that all writes were committed on the primary server, and all reads were from the primary server. For Cassandra, the effect is that all writes were committed on a majority quorum at each of the three sub-clusters, while a read required a majority quorum only on the local sub-cluster. For Riak, the effect was to require a majority quorum on the entire nine-node cluster for both write operations and read operations.

TABLE I Settings for representative production configuration

Database	Write Options	Read Options
MongoDB	Primary Acknowledged	Primary Preferred
Cassandra	EACH_QUORUM	LOCAL_QUORUM
Riak	quorum	Quorum

The throughput performance for the representative production configuration for each of the workloads is shown in Figs. 2, 3, and 4.

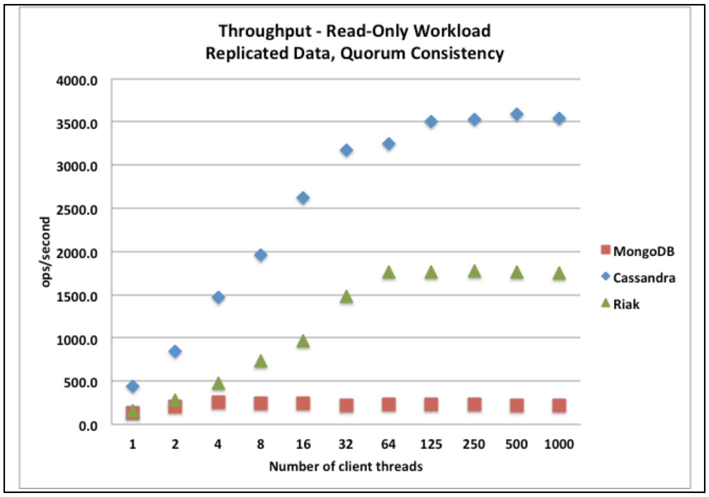


Fig. 2 Throughput, Representative Production Configuration, Read-Only Workload (higher is better)

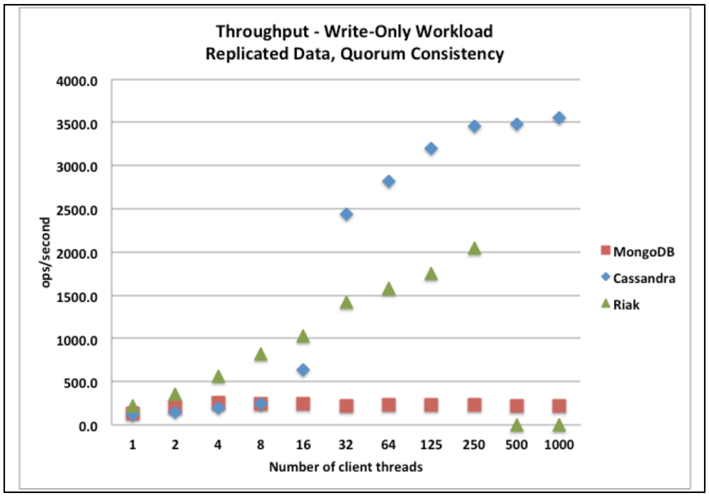


Fig. 3 Throughput, Representative Production Configuration, Write-Only Workload

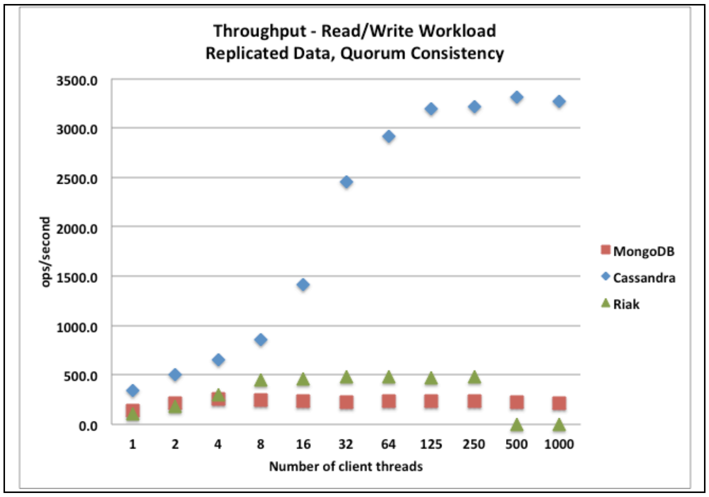


Fig. 4 Throughput, Representative Production Configuration, Read/Write Workload

In all cases, Cassandra provided the best overall performance, with read-only workload performance roughly comparable to the single node configuration, and write-only and read/write workload performance slightly better than the single node configuration. This implies that, for Cassandra, the performance gains that accrue from decreased contention for disk I/O and other per node resources (compared to the single node configuration) are greater than the additional work of coordinating write and read quorums across replicas and data centers. Furthermore, Cassandra's "data center aware" features provide some separation of replication configuration from sharding configuration. In this test configuration, this allowed a larger portion of the read operations to be completed without requiring request coordination (i.e. peer-to-peer proxying of the client request), compared to Riak.

Riak performance in this representative production configuration is better than the single node configuration. In test runs using the write-only workload and the read/write workload, our Riak client had insufficient socket resources to execute the workload for 500 and 1000 concurrent sessions. These data points are hence reported as zero values in Figs. 3 and 4. We later determined that this resource exhaustion was due to ambiguous documentation of Riak's internal thread pool configuration parameter, which creates a pool for *each* client session and not a pool shared by *all* client sessions. After determining that this did not impact the results for one through 250 concurrent sessions, and given that Riak had qualitative capability gaps with respect to our strong consistency requirements (discussed below), we decided not to re-execute the tests for those data points.

MongoDB performance is significantly lower here than the single node configuration. Two factors influenced the MongoDB results. First, the representative production configuration is sharded, which introduces the router and configuration nodes into the MongoDB deployment architecture. The router node proxies each request to the appropriate shard, based on key mapping information contained in the configuration node. In our tests, the router node became a performance bottleneck. Figs. 5 and 6 show read and write operation latency for the read/write workload, with nearly constant average latency for MongoDB as the number of concurrent sessions is increased, which we attribute to saturation of the rapid saturation of the router node.

The second factor affecting MongoDB performance is the interaction between the sharding scheme used by MongoDB and the write-only and read/write workloads that we used. Both Cassandra and Riak use a hash-based sharding scheme, which provides a uniformly distributed mapping from the range of keys onto the physical nodes. In contrast, MongoDB used a range-based sharding scheme with rebalancing[1].

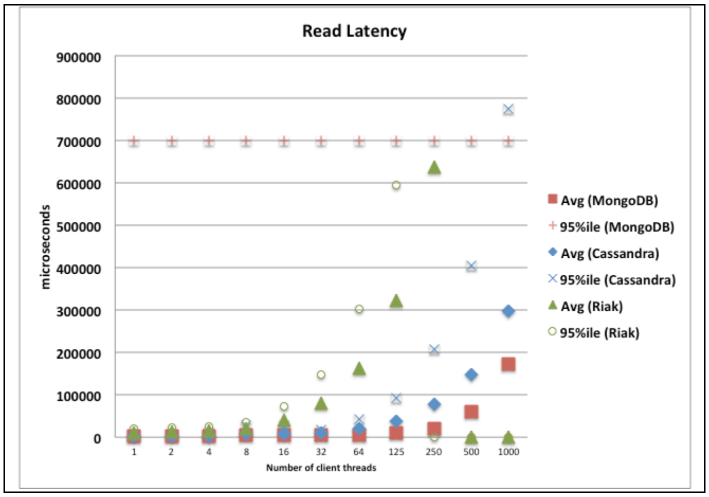


Fig. 5 Read Latency, Representative Production Configuration, Read/Write Workload

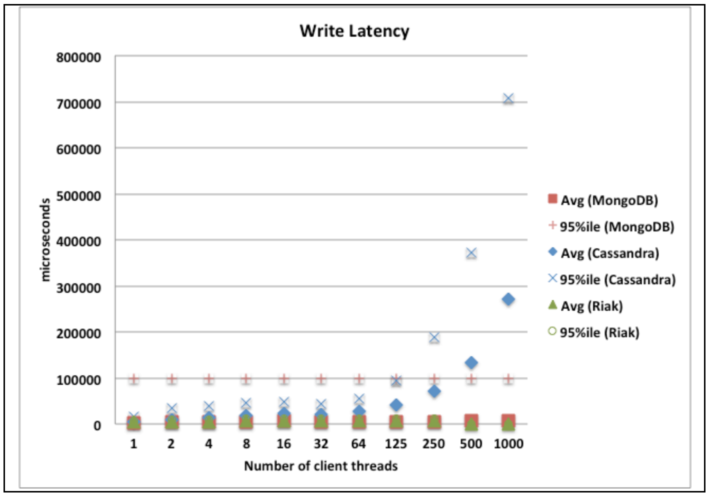


Fig. 6 Write Latency, Representative Production Configuration, Read/Write Workload

The use cases for our write-only and read/write workloads generated a monotonically increasing sequential key for new records to be written, which caused all write operations to be directed to the same shard, since all of the write keys mapped into the space stored in that shard. This key generation approach is typical (in fact, many SQL databases have "autoincrement" key types that do this automatically), but in this case, it concentrates the write load for all new records in a single node and thus negatively impacts performance. A different indexing scheme was not available to us, as it would impact other systems that our customer operates. (We note that MongoDB introduced hash-based sharding in v2.4, after our testing had concluded.)

Our tests also measured latency of read and write operations. While Cassandra achieved the highest overall throughput, it also delivered the highest average latencies. For example, at 32 client connections, Riak's read operation latency was 20% of Cassandra (5x faster), and MongoDB's write operation latency was 25% of Cassandra's (4x faster). Figs. 5

[1] http://docs.mongodb.org/v2.2/core/sharded-clusters/

and 6 show average and 95th percentile latencies for each test configuration.

B. Performance Evaluation – Eventual Consistency

Finally we report performance results that quantify the performance cost of strong replica consistency. These tests were limited to the Cassandra and Riak databases – the performance of MongoDB in the representative production configuration was such that no additional characterization of that database was warranted for our application. These tests used a combination of write and read operation settings that resulted in eventual consistency, rather than the strong consistency settings used in the tests described above. Again, each of the databases offered slightly different options. The selected options are summarized in TABLE II. The effect of these settings for both Cassandra and Riak was that writes were committed on one node (with replication occurring after the operation was acknowledged to the client), and read operations were executed on one replica, which may or may not return the latest value written.

TABLE II Settings for eventual consistency configuration

Database	Write Options	Read Options
Cassandra	ONE	ONE
Riak	noquorum	noquorum

For Cassandra, at 32 client sessions, there is a 25% reduction in throughput moving from eventual to strong consistency. Figure 7 shows throughput performance for the read/write workload on the Cassandra database, comparing the representative production configuration with the eventual consistency configuration.

The same comparison is shown for Riak in Figure 8. Here, at 32 client sessions, there is only a 10% reduction in throughput moving from eventual to strong consistency (As discussed above, test client configuration issues resulted in no data recorded for 500 and 1000 concurrent sessions.)

In summary, the Cassandra database provided the best throughput performance, but with the highest latency, for the specific workloads and configurations tested here. We attribute this to several factors. First, hash-based sharding spread the request and storage load better than MongoDB. Second, Cassandra's indexing features allowed efficient retrieval of the most recently written records, particularly compared to Riak. Finally, Cassandra's peer-to-peer architecture and data center aware features provide efficient coordination of both read and write operations across replicas and data centers.

VII. Data Model Mapping Results

Throughout the prototype design and development, we developed a set of findings that are qualitative. Here we report on these qualitative findings in the area of alignment of our data model with the capabilities provided by each database.

The most significant data modeling challenge was the representation of the one-to-many relation from patient to lab results, coupled with the need to efficiently access the most-recently written lab results for a particular patient.

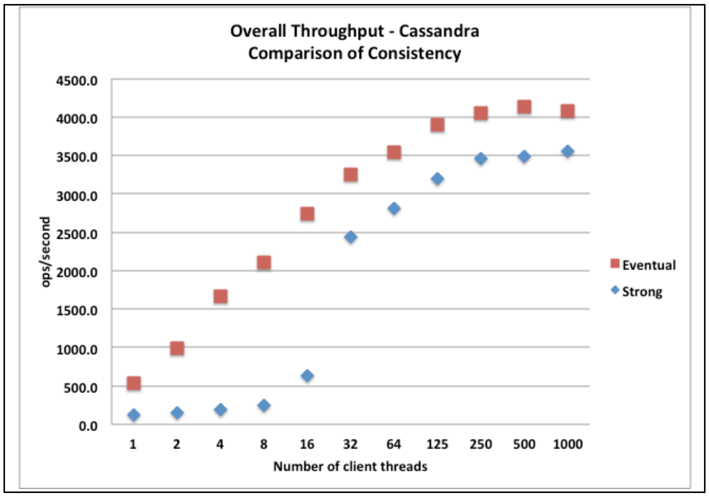


Fig. 7 Cassandra – Comparison of strong and eventual consistency

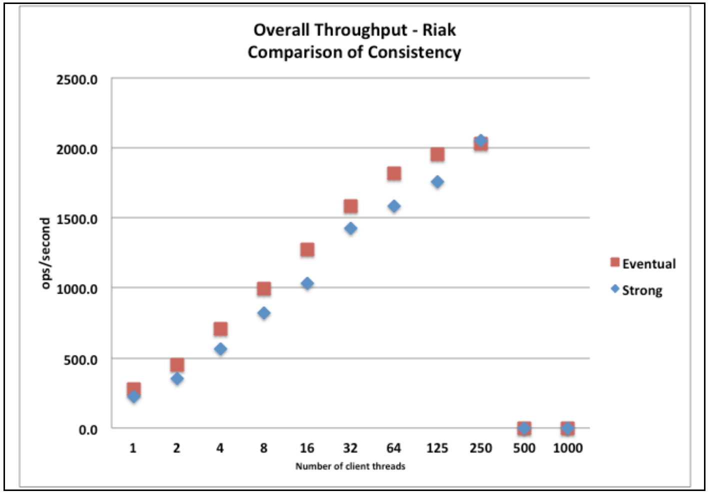


Fig. 8 Riak – Comparison of strong and eventual consistency

Zola has analyzed the various approaches and tradeoffs of representing the one-to-many relation in MongoDB [25]. We used a composite index of ⟨Patient ID, Observation ID⟩ for lab result records, and also indexed by the lab result date-time stamp. This allowed efficient retrieval of the most recent lab result records for a particular patient.

A similar approach was used for Cassandra. Here we used a composite index of ⟨PatientID, lab result, date-time stamp⟩. This caused the result set returned by the query to be sorted by the server, making it efficient to find and filter the most recent lab records for a particular patient.

Representing the one-to-many relation in Riak was more complicated. Riak's key-value data model provides the capability to retrieval a value, given a unique key. Riak also provides a "secondary index" capability that allows record retrieval when the key is not known, however each server node in the cluster stores only the secondary indexes for the portion of the records that are managed by the node. When an operation requests all records with a particular secondary index value, the request coordinator must perform a "scatter-gather", asking all storage node for records with the desired secondary index value, waiting for all nodes to respond, and then sending the list of keys back to the requester. The requester must then make a second database request with the list of keys, in order to retrieve the record values.

The latency of the "scatter-gather" to locate records, and the need for two request/response round trips had a negative impact on Riak's performance for our data model. Furthermore, there is no mechanism in Riak for the server to filter and return only the most recent observations for a patient. All observations must be returned to the client, and then sorted and filtered. We attempted to de-normalize further, by introducing a data set where each record contained a list of the most recently written observations for each patient. However, this required an atomic read-modify-write of that list every time a new observation is added for the patient, and that capability was not supported in Riak version 1.4.

We also performed a systematic categorization of the major features available in the three databases for issuing queries. For each feature, we classified the capabilities of each database against a set of criteria so that they can be directly compared. TABLE III shows the results of this comparison.

This comparison allows our customer to evaluate the databases against their specific needs beyond the scope of this evaluation, both for runtime and software development (e.g. programming language support, data type support). Both these sets of requirements are important factors in any technology adoption decision, and must be weighted appropriately to best satisfy organizational requirements.

Objectively, MongoDB and Cassandra both provided a relatively straightforward data model mapping and both provided the strong consistency needed for our customer's EHR application. Subjectively, the data model mapping in MongoDB was more transparent than the use of the Cassandra Query Language (CQL), and the indexing capabilities of MongoDB were a better fit for this application.

VIII. Lessons Learned

We present our lessons learned in two broad categories. The first set are issues that arose from the essential complexity of evaluating NoSQL products. The second set are issues that arose from the accidental complexity of the available tools and technologies.

A. Essential Issues

1) Defining selection criteria

NoSQL technology selection is an architecture decision that must be made early in the design cycle, and is difficult and expensive to change [6]. The selection must be made in a setting where the problem definition may be incomplete, and the solution space is large and rapidly changing as the open source landscape continues to evolve.

Our experience was that the decision drivers were the size and growth rate of the data (number of records and record size), the complexity of the data model including key relations and navigations, operational environment including system management practices and tools, and user access patterns including operation mix, queries, and number of concurrent users. We found that using quality attribute scenarios to elicit these requirements, followed by clustering and prioritization to identify "go/no-go" criteria, was an effective approach to defining selection criteria.

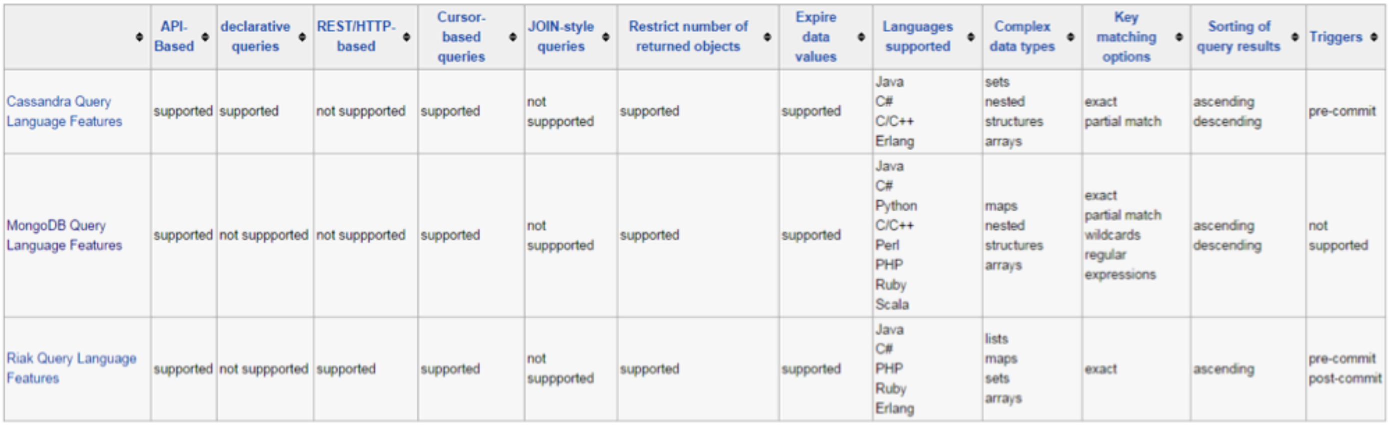
2) Validating quantitative criteria

Quantitative selection criteria, with hard "go/no-go" thresholds, were problematic to validate through prototyping. There are a large number of tunable parameters in the infrastructure, operating system, and database product. While the final architecture design must include that tuning, the testing space can quickly explode during selection. We found it useful to frame the performance criteria in terms of the shape of the performance curve. For example, does throughput increase linearly with increasing load throughout the range of interest? Understanding the sensitivities and trade offs in a product's capabilities may be sufficient to make a selection, and also provides valuable information to make downstream architecture design decisions regarding the selected product.

3) Screening candidate products to prototype

We used architecturally significant requirements to perform a manual survey of product documentation to identify viable candidates for prototyping. The manual survey process was slow and inefficient – as noted above, the solution space is large and rapidly changing. We began to collect and aggregate product feature and capability information into a queryable, reusable knowledge base, which included general quality

TABLE III Comparing the Data Querying Capabilities for Cassandra, MongoDB, and Riak



	API-Based	declarative queries	REST/HTTP-based	Cursor-based queries	JOIN-style queries	Restrict number of returned objects	Expire data values	Languages supported	Complex data types	Key matching options	Sorting of query results	Triggers
Cassandra Query Language Features	supported	supported	not suppported	supported	not suppported	supported	supported	Java C# C/C++ Erlang	sets nested structures arrays	exact partial match	ascending descending	pre-commit
MongoDB Query Language Features	supported	not suppported	not suppported	supported	not suppported	supported	supported	Java C# Python C/C++ Perl PHP Ruby Scala	maps nested structures arrays	exact partial match wildcards regular expressions	ascending descending	not supported
Riak Query Language Features	supported	not suppported	supported	supported	not suppported	supported	supported	Java C# PHP Ruby Erlang	lists maps sets arrays	exact	ascending	pre-commit post-commit

attribute scenarios as templates for concrete scenarios, and linked the quality attribute scenarios to particular product features. This knowledge base was reused successfully for later projects, and is an area for further research.

4) Tradeoff between evaluation cost and fidelity

The selection process must balance cost (in time and resources) with fidelity and measurement precision. This is essential for any COTS selection, but the NoSQL context creates challenges as the solution space changes rapidly. During the course of our evaluation, each of the candidate products released at least one new version that included changes to relevant features, so a lengthy evaluation process is likely to produce results that are not relevant or valid. Furthermore, if a public cloud infrastructure is used to support the prototyping and measurement, then changes to that environment can impact results. For example, during our testing process, Amazon changed standard instance types offered in EC2. Our recommendation is to perform prototyping and measurement for just two or three products, in order to complete quickly and deliver valid and relevant results.

B. Accidental Issues

1) Tradeoff between manual testing and automation

We performed all prototyping and measurement using the Amazon cloud, which proved essential for efficient management and execution of the tests. Our peak utilization was over 50 concurrently executing server nodes (divided across several product configurations), which is more than can be efficiently managed in physical hardware environments.

We had a continual tension between using manual processes for server deployment and management, and automating some or all of these processes. Repeating manual tasks conflicts with software engineering best practices such as “don’t repeat yourself”[2], but in retrospect we think that the decision to always make slow forward progress, rather than stopping to automate, was appropriate. Organizations that already have a proven automation capability and expertise in place may reach a different conclusion. We did develop scripts to automate test execution and data collection, processing, and visualization. These tasks were performed frequently, had many steps, and needed to be repeatable.

2) Initial database loading

Evaluation of big data systems requires that the database contain a large data set. Our use cases required the database to be populated before running the workloads. We found that bulk or batch loading requires special attention. Each database product had specific recommendations and special APIs for this function. In some cases (i.e., MongoDB), recommendations like “pre-splitting” the data set significantly improved bulk load performance. In other cases, we found that following the recommendations was necessary to avoid failures due to resource exhaustion in the database server during the load processing. We recommend that if bulk load is not one of your selection criteria, then take a brute force approach to load the data once, and then use database backups, or virtual machine or storage volume snapshots to return to the initial state as needed.

3) Deleting records at completion of a test

All of our tests that performed write operations ended the test by restoring the database to its initial state. We found that deleting records in most NoSQL databases is very slow, taking as much as 10 times longer than a read or write operation. In retrospect, we would consider using snapshots to restore state, rather than cleaning up using delete operations.

4) Measurement framework

It is critical that you understand your measurement framework. Although YCSB has become the *de facto* standard for NoSQL database characterization, the 95th and 99th percentile measurements that it reports are only valid under certain latency distribution conditions. The YCSB implementation could be modified to extend the validity of those measurements to a broader range of latencies, or alternative metrics can be used for selection criteria.

IX. Further Work and Conclusions

NoSQL database technology offers benefits of scalability and availability through horizontal scaling, replication, and simplified data models, but the specific implementation must be chosen early in the architecture design process.

Ultimately, technical capabilities are just one input to the technology selection decision. Non-technical factors such as development and operational cost, schedule, risk, alignment with organizational standards are also considered, and may have more influence on the final decision. However, a rigorous technical evaluation, based on prototyping and measurement, provides important information to assess both technical and non-technical considerations.

We have described a systematic method to perform this technology evaluation in a context where the solution space is broad and changing fast, and the system requirements may not be fully defined. Our method evaluates the products in the specific context of use, starting with elicitation of quality attribute scenarios to capture key architecture drivers and selection criteria. Next, product documentation is surveyed to identify viable candidate technologies, and finally, rigorous prototyping and measurement is performed on a small number of candidates to collect data to make the final selection.

We described the execution of this method to evaluate NoSQL technologies for an electronic healthcare system, and present the results of our measurements of performance, along with a qualitative assessment of alignment of the NoSQL data model with system-specific requirements. We presented lessons learned from our application of the selection method, and from our execution of the prototyping and measurements.

Our experience identified the benefits of having a trusted knowledge base that can be queried to discover the features and capabilities of particular NoSQL products, and accelerate the initial screening to identify viable candidate products for a particular set of quality attribute scenario requirements. This is an area for further research.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-

[2] http://c2.com/cgi/wiki?DontRepeatYourself

0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. T-Check^{SM}. DM-0002078.

References

[1] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," in *Proc. 14th Int'l Conf. on Extending Database Tech.*, 2011, pp. 530--533.

[2] M. Seltzer. Beyond Relational Databases. *Communications of the ACM*, 51(7), 2008.

[3] F. Chang, J. Dean, S. Ghemawat, et al., "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. on Computing Systems,* vol. 26, no. 2, 2008.

[4] G. DeCandia, D. Hastorun, M. Jampani, et al., "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. Twenty-first ACM SIGOPS Symp. on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007, pp. 205-220. doi: 10.1145/1294261.1294281

[5] P. J. Sadalage and M. Fowler, *NoSQL Distilled.* Addison-Wesley Professional, 2012.

[6] I. Gorton and J. Klein, "Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems," *IEEE Software,* vol. PP, no. 99, 18 March 2014. doi: 10.1109/MS.2014.51

[7] S. Comella-Dorda, J. Dean, G. Lewis, et al., "A Process for COTS Software Product Evaluation." Software Engineering Institute, Technical Report, CMU/SEI-2003-TR-017, 2004.

[8] J. Zahid, A. Sattar, and M. Faridi. "Unsolved Tricky Issues on COTS Selection and Evaluation." *Global Journal of Computer Science and Technology* 12.10-D (2012).

[9] Becker, C., Kraxner, M., Plangg, M., & Rauber, A. Improving decision support for software component selection through systematic cross-referencing and analysis of multiple decision criteria. In *Proc. 46th Hawaii International Conference on System Sciences (HICSS)*, 2013 , pp. 1193-1202.

[10] I. Gorton, A. Liu, and P. Brebner, "Rigorous evaluation of COTS middleware technology," *Computer,* vol. 36, no. 3, pp. 50-55, 2003.

[11] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," in *Proc. 9th Int'l Conf. on Very Large Data Bases (VLDB '83)*, 1983, pp. 8-19.

[12] Anon, D. Bitton, M. Brown, et al., "A Measure of Transaction Processing Power," *Datamation,* vol. 31, no. 7, pp. 112-118, April 1985.

[13] B. F. Cooper, A. Silberstein, E. Tam, et al., "Benchmarking Cloud Serving Systems with YCSB," in *Proc. 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 143-154. doi: 10.1145/1807128.1807152

[14] S. Patil, M. Polte, K. Ren, et al., "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *Proc. 2nd ACM Symp. on Cloud Computing (SOCC '11)*, 2011, pp. 9:1--9:14. doi: 10.1145/2038916.2038925

[15] D. Nelubin and B. Engber, "Ultra-High Performance NoSQL Benchmarking: Analyzing Durability and Performance Tradeoffs." Thumbtack Technology, Inc., White Paper, 2013.

[16] Datastax, "Benchmarking Top NoSQL Databases." Datastax Corporation, White Paper, 2013.

[17] V. Abramova and J. Bernardino, "NoSQL Databases: MongoDB vs Cassandra," in *Proc. Int'l C* Conference on Computer Science and Software Engineering (C3S2E '13)*, 2013, pp. 14-22. doi: 10.1145/2494444.2494447

[18] A. Floratou, N. Teletia, D. J. DeWitt, et al., "Can the Elephants Handle the NoSQL Onslaught?," *Proc. VLDB Endowment,* vol. 5, no. 12, pp. 1712-1723, 2012.

[19] G. A. Lewis and L. Wrage, "A Process for Context-Based Technology Evaluation." Carnegie Mellon Software Engineering Institute, Technical Note, CMU/SEI-2005-TN-025, 2005.

[20] E. Dede, M. Govindaraju, D. Gunter, et al., "Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis," in *Proc. 4th ACM Workshop on Scientific Cloud Computing (Science Cloud '13)*, 2013, pp. 13-20. doi: 10.1145/2465848.2465849

[21] M. R. Barbacci, R. J. Ellison, A. J. Lattanze, et al., "Quality Attribute Workshops (QAWs)." Software Engineering Institute, Technical Report, CMU/SEI-2003-TR-016, 2003, http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=6687.

[22] Y. Liu, I. Gorton, L. Bass, C. Hoang, & S. Abanmi. MEMS: a method for evaluating middleware architectures. In Proceedings of the Second international conference on Quality of Software Architectures (QoSA'06), 2006, Springer-Verlag, Berlin, Heidelberg, pp. 9-26.

[23] A. Liu and I. Gorton. 2003. Accelerating COTS Middleware Acquisition: The i-Mate Process. IEEE Software. 20, 2 (March 2003), 72-79.

[24] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM,* vol. 56, no. 2, pp. 74-80, February 2013. doi: 10.1145/2408776.2408794

[25] W. Zola. *6 Rules of Thumb for MongoDB Schema Design: Part 1* [Online]. http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1 (Accessed 18 Sep 2014).