Application-Specific Evaluation of NoSQL Databases

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— The selection of a particular NoSQL database for use in a big data system imposes a specific distributed software architecture and data model, making the technology selection difficult to defer and expensive to change. This paper reports on the selection of a NoSQL database for use in an Electronic Healthcare Record system being developed by a large healthcare provider. We performed application-specific prototyping and measurement to identify NoSQL products that fit data model and query use cases, and meet performance requirements. We found that database throughput varied by a factor of 10, read operation latency varied by a factor of 5, and write latency by a factor of 4 (with the highest throughput product delivering the highest latency). We also found that the throughput for workloads using strong consistency was 10-25% lower than workloads using eventual consistency. We conclude by reflecting 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

At the heart of many big data systems are "NoSQL" database management systems that are simpler than traditional relational databases and provide higher scalability and availability [1]. These databases 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 of servers. Each of these products implements a different data model and query language, as well as specific mechanisms to achieve distributed data consistency and availability.

When a big data system uses a particular database, the data, consistency and distribution models imposed by the database have a pervasive impact on the design of the associated applications **Error! Reference source not found.**. 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.

Commercial off-the-shelf (COTS) software selection has been extensively studied in software engineering [3][4][5]. In complex technology landscapes with multiple competing products, developers must balance the cost and speed of the selection process against the fidelity of the analysis [6]. 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.

There are several unique challenges that make selection of NoSQL databases for use in big data applications a particularly hard problem:

- This is an early architecture decision that must be made with inevitably incomplete requirements.
- Capabilities and features vary widely across NoSQL databases and performance is very sensitive to how a product's data model and query features match application needs, making generalized comparisons difficult.
- Production-scale prototypes, with hundreds of servers, multi-terabyte data sets, and thousands or millions of clients, are usually impractical.
- The solution space is changing rapidly, with new products 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 considering the use of NoSQL databases for an Electronic Health Record (EHR) system.

The next section provides details of the project context and technology evaluation approach. This is followed in Section III by a discussion of the prototype design and configuration. Section IV presents the performance test results. We conclude with a reflection on lessons learned during this project.

II. EHR CASE STUDY

A. Project Context

Our customer was a large healthcare provider developing a new EHR system. This system supports healthcare delivery for over 9,000,000 patients in more than 100 facilities across the globe. The data growth rate is more than one terabyte per month, and all data must be retained for 99 years.

NoSQL technologies were considered attractive candidates for two specific uses, namely:

- the primary data store for the EHR system

978-1-4673-7278-7/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataCongress.2015.83

- a local cache at each site to improve request latency and availability

This will replace an existing system that uses "thick client" applications running at sites around the world, all connected to a centralized relational database, so the customer wanted to characterize performance with hundreds of concurrent database sessions.

The customer was familiar with RDMS technology for these use cases, but had no experience using NoSQL, so we were directed to focus our evaluation only on NoSQL technology.

B. Evaluation Approach

The approach is inspired by previous work on middleware evaluation [6][7] and customized to address the characteristics of big data systems. The basic main steps are depicted in Fig. 1 and outlined below.

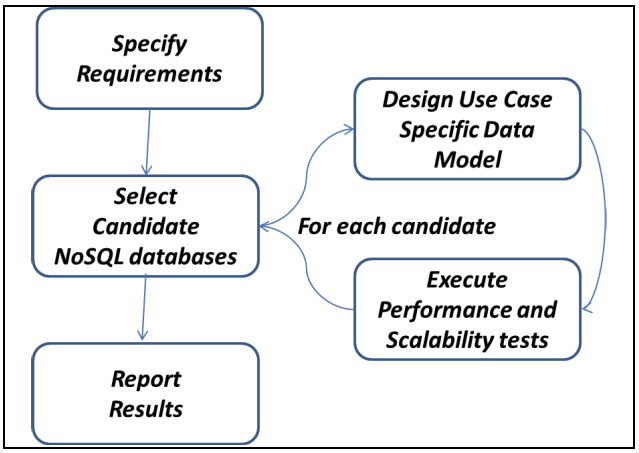

Fig. 1. Lightweight Evaluation and Prototyping for Big Data (LEAP4BD)

1) Specify Requirements

We used a stakeholder workshop to elicit key functional and quality attribute requirements to frame the evaluation. These key requirements were:

Performance/Scalability: The main quantitative requirements were to replicate data across geographically distributed data centers, 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: Healthcare systems have common logical data models and query patterns that need to be supported by a NoSQL database. This required us to evaluate the specific data modeling and query features for each product, including capabilities to maintain replica consistency in a distributed deployment.

We next helped the customer define two primary use cases for the EHR system. These drove the evaluation that we performed in subsequent steps in the project. The first use case was to read recent medical test results for a single patient, is a core function used to populate the user interface whenever a clinician selects a new patient. The second use case was achieving strong replica consistency when a new medical test result is written for a patient, so that all clinicians using the EHR to make patient care decisions will see the same information, whether they are at the same site as the patient, or providing telemedicine support from another location.

2) 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, and so we selected one NoSQL database from each category to investigate in detail. We subsequently ruled out graph databases, as none provided the horizontal partitioning required for this customer's application. We chose Riak, Cassandra and MongoDB as the three candidates, based on product maturity and availability of enterprise support.

3) Design and Execute Performance Tests

In order to make an "apples to apples" comparison of the databases that were evaluated, we defined and performed a systematic test procedure. Based on the use cases defined during the requirements step, we:

- Defined and implemented a consistent test environment, which included server platform, test client platform, and network topology.
- Mapped the logical model for a patient's medical test history onto 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 product 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.

This enabled us to produce a consistent set of test results that assess the likely performance and scalability of each database for this customer's EHR system. The details of the environment and test design are presented in the next section.

III. 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/).

In Section IV, we report performance results for two database server configurations: Single node server, and a nine-node configuration that was representative of a production deployment. Executing on a single node allowed us to validate our test environment for each database. The nine-node cluster was configured to represent a geographically distributed deployment across three data centers. The data set was sharded across three nodes, and then replicated to two additional groups of three nodes each. This was achieved using MongoDB's primary/secondary feature, and Cassandra's data center aware distribution feature. Riak did not directly 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.

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

B. Mapping the data model

Most of the prototyping effort was spent mapping the application-specific logical data model onto the particular data model, indexing, and query language capabilities of each database to be tested.

We used the HL7 Fast Healthcare Interoperability Resources (FHIR)[1] as the logical data model for our analysis and prototyping. The set of all test results for a patient, were modeled using the "FHIR Patient Resources" (e.g., demographic information such as names, addresses, and telephone numbers) along with "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.

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. Zola has analyzed the various approaches and tradeoffs of representing the one-to-many relation in MongoDB [11]. 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 retrieve the most recent lab records for a particular patient.

In Riak, representing the one-to-many relation 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 to avoid a full scan when the key is not known. However, each node in the cluster stores only the secondary indices for those shards stored by the node. A query to match a secondary index value causes the request coordinator to perform a "scatter-gather", asking each node for records with the requested secondary index value, waiting for all nodes to respond, and then sending the list of keys for the matching records back to the requester. The requester must then make a second request with the list of keys, to retrieve the record values.

The latency of the "scatter-gather" to locate records and the need for two round trips to retrieve the records had a negative impact on Riak's performance for our data model. Since there is no mechanism in Riak for the server to filter and return only the most recent observations for a patient, all matching records must be returned and then sorted and filtered by the client.

MongoDB and Cassandra both provided a relatively straightforward data model mapping and both provided the strong replica consistency needed for this application. The data model mapping for MongoDB seemed more transparent than the use of the Cassandra Query Language (CQL), and the indexing capabilities of MongoDB were a better fit for this application.

C. Generate and load data

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 0 to 20, with an average of 7.

D. Create load test client

We used the YCSB framework [8] as the foundation for the test client, to manage test execution and test measurement. For test execution, we replaced YCSB's very simple default data models, data sets, and queries with implementations specific to our use case data and requests.

YCSB's built-in capabilities allow specification of the total number of operations and the mix of read and write operations in a workload. Our customer specified that the typical workload for the EHR system was 80% read and 20% write operations. For this operation mix, we implemented 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.

In order to investigate using the NoSQL technology as a local cache (described in §II.A, above), we implemented a write-only workload that represented a daily operation to load a local cache from a centralized primary data store with records for patients with scheduled appointments for that day. We also implemented a read-only workload that

[1] http://www.hl7.org/implement/standards/fhir/

represented flushing the cache back to the centralized primary data store.

The YCSB measurement framework measures *operation latency* as the time from when the request is sent to the database until the response is received back from the database. The YCSB reporting framework aggregates latency measurements separately for read and write operations. Latency distribution is a key scalability metric for big data systems [9][10], so we recorded both average and 95th percentile values.

We extended YCSB to report *Overall Throughput*, in operations per second. This was the total number of operations performed (reads plus writes) divided by the workload execution time (from the start of the first operation to the completion of the last operation in the workload execution, and not including initial setup and final cleanup times).

E. Define and execute test scripts

For each database and configuration, every workload was run three times, to minimize the impact of transient events in the cloud infrastructure. The standard deviation of the throughput for any three-run set never exceeded 2% of the average.

YCSB can use multiple execution threads to create concurrent client sessions, so for each of the three test 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. Post-processing of test results averaged measurements across the three runs for each thread count.

NoSQL databases are not typically designed to operate with a large number of concurrent database client sessions. Usually, clients connect to a web server tier and/or an application server tier, which aggregates the client operations on the database using a pool of roughly 16-64 concurrent sessions. However, since this customer was modernizing a system that used thick clients with direct database connections, they wanted to understand the feasibility of retaining the thick client architecture.

Since there were multiple concurrent connections to the database, we had to define how these were distributed across the server nodes. MongoDB uses a centralized router node, so all clients connected to that 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 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.

IV. Performance and Scalability Test Results

We report here on results for the nine-node configuration that reflected a typical production system (described in §III.A above). Other tested configurations included running on a single server. The single-node configuration's availability and scalability limitations make it unfeasible 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 particular database to its distributed configuration, to provide insights into the efficiency of distributed coordination mechanisms and guide tradeoffs to scale up by adding more nodes versus using faster nodes with more storage.

This EHR application required strong replica consistency. These results are reported first, below. This is followed by a comparison of strong replica consistency to eventual consistency.

A. Performance Evaluation – Strong Consistency

The database configuration options to achieve strong replica consistency are summarized in TABLE I. . For MongoDB, these settings cause all writes to be committed on the primary server, and all reads are from the primary server. For Cassandra, the effect is that all writes are committed on a quorum formed on each of the three sub-clusters, while a read required a quorum only on the local sub-cluster. For Riak, the effect is to require a 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.

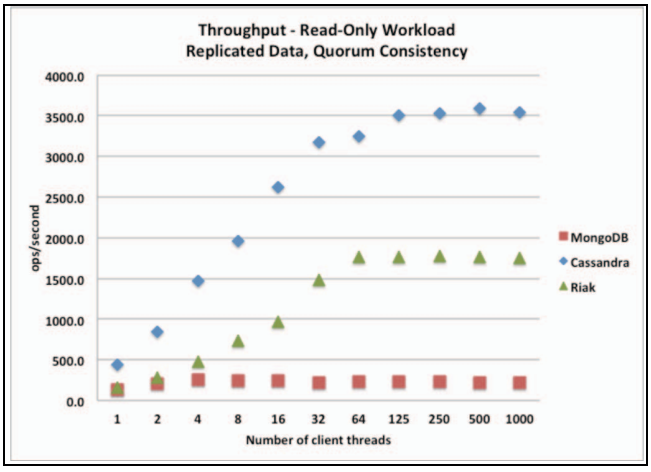

Fig. 2. Throughput, Representative Production Configuration, Read-Only Workload (higher is better)

In every case, Cassandra provided the best overall performance (peaking at approximately 3500 operations per second), with read-only workload performance approximately 10% better than the single node configuration, and write-only and read/write workload performance approximately 25% higher than the single node

configuration. In moving from single node to a distributed configuration, we gain performance from decreased contention for storage I/O and other per-node resource. We also lose performance, due to the additional work of coordinating write and read quorums across replicas and data centers. For Cassandra, the gains exceeded the losses, resulting in net higher performance in the distributed configuration.

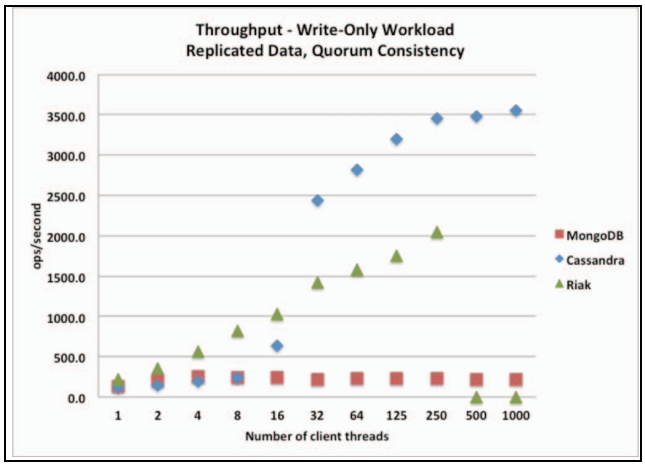

Fig. 3. Throughput, Representative Production Configuration, Write-Only Workload

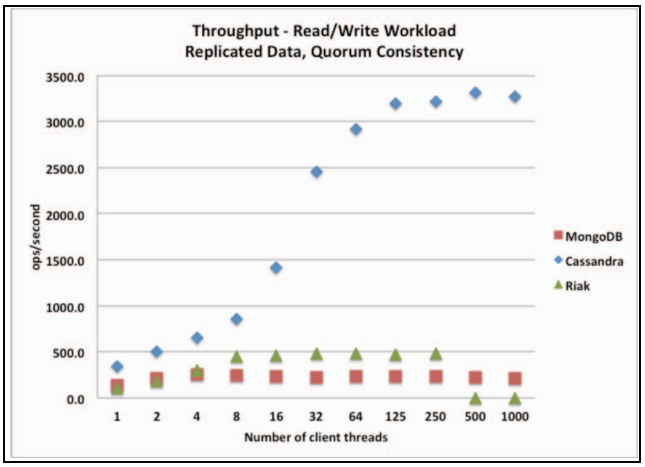

Fig. 4. Throughput, Representative Production Configuration, Read/Write Workload

Furthermore, Cassandra's "data center aware" features provide some separation of replication configuration from sharding configuration. In these tests, compared to Riak, 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),.

Riak performance in this distributed configuration is approximately 2.5x 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's single node configuration performance was nearly 8x better than the distributed configuration. We attribute this to two factors: First, the distributed configuration is sharded, which introduces the router and configuration nodes into the MongoDB deployment architecture. The router proxies each request to the appropriate shard, using the key mapping stored 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 rapid saturation of the single 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[2].

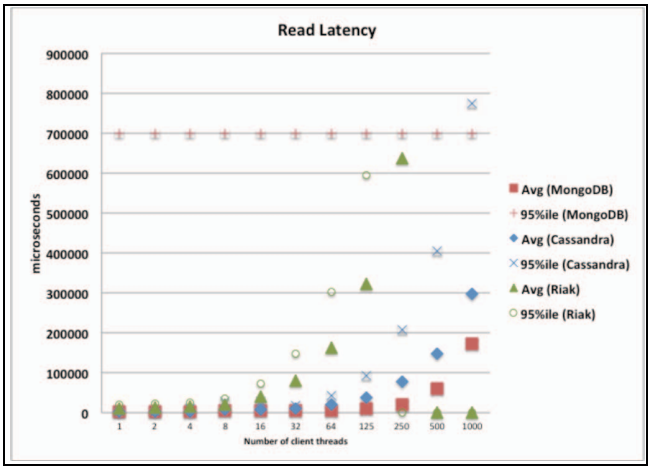

Fig. 5. Read Latency, Representative Production Configuration, Read/Write Workload

Our workloads generated a monotonically-increasing 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 range stored in that shard. This is

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

a typical key generation approach (e.g., the SQL "autoincrement" key types), but in this case, it focuses the write load for all new records onto 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.)

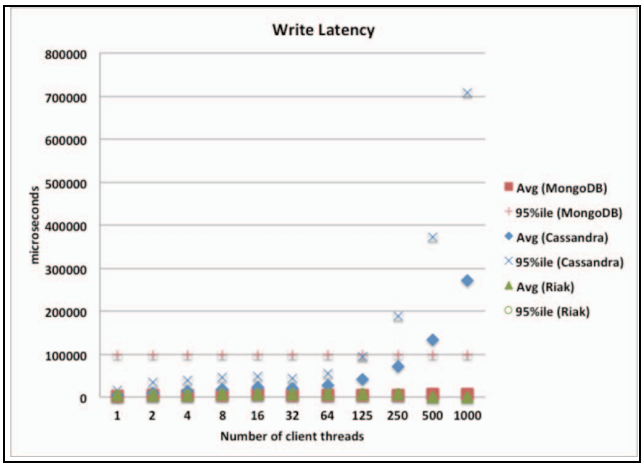

Fig. 6. Write Latency, Representative Production Configuration, Read/Write Workload

Our tests also measured latency of read and write operations. While Cassandra achieved the highest overall throughput (approximately 3500 operations per second), it also delivered the highest average latencies (indicative of high internal concurrency in request processing). 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 and 6 show average and 95th percentile latencies for each test configuration.

B. *Performance Evaluation – Eventual Consistency*

We also performed tests to quantify the performance cost of strong replica consistency, compared to eventual 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. The selected write and read options to achieve eventual consistency 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 going from eventual consistency 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. (As discussed above, test client configuration issues resulted in no data recorded for 500 and 1000 concurrent sessions.)

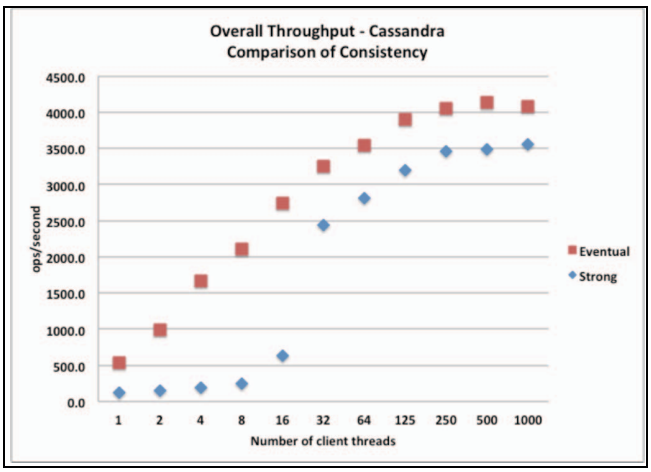

Fig. 7. Cassandra – Comparison of strong and eventual consistency

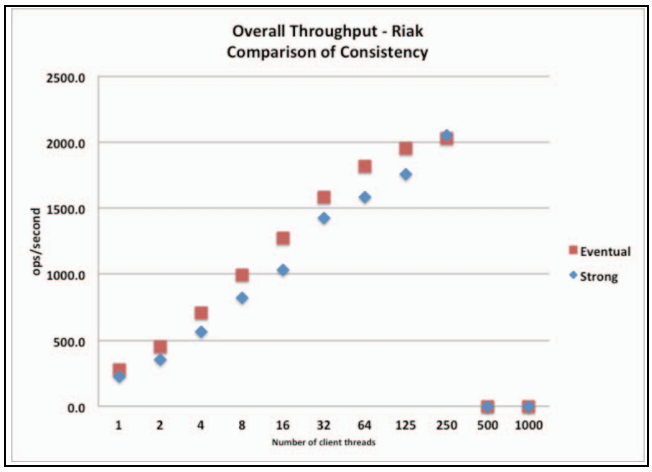

Fig. 8. Riak – Comparison of strong and eventual consistency

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.

V. LESSONS LEARNED

Product evaluation of NoSQL databases presents a number of challenges that we had to address in the course of this project. Our lessons learned fall in two broad categories: The first category includes issues related the essential complexity of evaluating NoSQL products, and the second category includes issues that arose from the accidental complexity of the available tools and technologies.

A. Essential Issues

1) Defining selection criteria

This technology selection decision must be made early in the design cycle, and may be difficult and expensive to change **Error! Reference source not found.**. The selection must be made in a context where the problem definition may not be complete, and the solution space is large and rapidly changing as the open source landscape continues to evolve.

We found that principal decision drivers were the size and growth rate of the data (number of records and record size), the complexity of the data model including the relations and navigations to support the use cases, the operational environment including system management practices and tools, and user access patterns including operation mix, queries, and number of concurrent users. We used quality attribute scenarios [12] to elicit these requirements, followed by clustering and prioritization. There were diverse stakeholder concerns, and identifying "go/no-go" decision criteria helped to focus the evaluation.

2) Configuration tuning

There are many configuration tuning parameters available, at the database, operating system, and EC2 level. We minimized changes to default configurations for two reasons. First, tuning can be a lengthy process, balancing interactions between settings within a layer and across layers. Second, our workload and data set were representative of, but not identical to, our production system and so optimization for our test workload would not necessarily apply to the production system. We found significant performance differences between products that would not be eliminated by tuning, and these were sufficient to make a selection.

3) Quantitative selection criteria

Quantitative selection criteria with hard thresholds were problematic to validate through prototyping. There are a many parameters that can be tuned to affect performance, in the infrastructure, operating system, and database product itself. While the final target system architecture 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, is there a linear increase in throughput as the load increases? If not, are there discontinuities or inflection points within the input range of interest? Understanding the sensitivities and trade offs in a product's capabilities can be sufficient to make a selection, and also provides valuable information to make downstream architecture design decisions regarding the selected product.

4) 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 earlier, 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 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.

5) Tradeoff between evaluation cost and fidelity

Any COTS selection process must balance cost (in time and resources) against fidelity (along dimensions such as data set size, cluster size, and exact configuration tuning), and the rapid changes in NoSQL technology exacerbate these issues. 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 limit prototyping and measurement to just two or three products, in order to finish quickly and produce results that are both valid and relevant in this evolving context.

B. Accidental Issues

1) Choosing between manual and automated testing

The prototyping and measurement reported here used the Amazon cloud, which enabled efficient management and execution of the tests. Our peak utilization was more than 50 concurrently executing server nodes (supporting several product configurations), which is more than can be efficiently managed in physical hardware environments.

We had constant tension between using manual processes for server deployment and management, and automating some or all of these processes. While repeating manual tasks goes against software engineering best practices such as "don't repeat yourself"[3], in retrospect we think that the decision to make slow, but constant, forward progress, rather than stopping to introduce automation, was appropriate. In organizations that have an automation capability and expertise in place may reach a different conclusion. We did automate test execution and data collection, processing, and visualization. These tasks were performed frequently, had many steps, and had to be repeatable.

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

2) Initial database loading

Evaluation of a big data system requires that the database under test contains a large data set. Our read-intensive use cases required populating the database test execution. We found that bulk or batch loading of NoSQL databases requires special attention; each database product had specific recommendations and special APIs for this function. For example, recommendations like "pre-splitting" the data set significantly improved bulk load performance (e.g., for MongoDB). In other cases, we found that following the recommendations was absolutely 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, consider taking 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 a very slow operation, 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 necessary to understand the measurement framework used in the test client. Although YCSB is the *de facto* standard for NoSQL database characterization, the 95th and 99th percentile measurements that it reports are incorrect 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.

VI. Further Work and Conclusions

Ultimately, technical capabilities and performance are just one input to a software technology selection decision. Non-technical factors such as development and operational cost, schedule, risk, and 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 for NoSQL database technology, in a context where the solution space is broad and changing fast, and the system requirements may not be fully defined. Our approach was to evaluate the products in the specific context of use, starting with elicitation of key requirements to capture 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 discussion of alignment of the NoSQL data model with system-specific requirements. We presented lessons learned from our experience on this project.

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-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. DM-0002241.

References

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

[2] I. Gorton and J. Klein, "Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems," *IEEE Software,* vol. 32, no. 3, pp. 78-85, May/June 2015. doi: 10.1109/MS.2014.51

[3] 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.

[4] 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).

[5] 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 Intl. Conf. on System Sciences (HICSS)*, 2013, pp. 1193-1202.

[6] Y. Liu, I. Gorton, L. Bass, C. Hoang, & S. Abanmi. "MEMS: a method for evaluating middleware architectures." In *Proc. 2nd Int'l Conf. on Quality of Software Architectures* (QoSA), 2006, pp. 9-26.

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

[8] 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

[9] G. DeCandia, D. Hastorun, M. Jampani, et al., "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007, pp. 205-220.

[10] 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

[11] 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).

[12] 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.