High-Volume Performance Test Framework using Big Data

Michael Yesudas
IBM Corporation
750 W. John Carpenter Frwy
Irving, TX 75039
United States of America
(+1) 469 549 8444
myesudas@us.ibm.com

Girish Menon S
IBM United Kingdom Limited
3 Furzeground Way, Stockley Park
Uxbridge, Middlesex, UB11 1EZ
United Kingdom
(+44) 208 867 8003
girishmenon@uk.ibm.com

Satheesh K Nair
IBM India Private Limited
Tower 'D', IBC Knowledge Park
Bannerghatta Road, Bangalore,
Karnataka, India
(+91) 803 090 6000
satheesht@in.ibm.com

ABSTRACT
The inherent issues with handling large files and complex scenarios cause the data-driven approach [1] to be rarely used for performance tests. Volume and scalability testing of enterprise solutions typically requires custom-made test frameworks because of the complexity and uniqueness of data flow. The generation, transformation and transmission of large sets of data pose a unique challenge for testing a highly transactional back-end system like the IBM Sterling Order Management (OMS). This paper describes a test framework built on document-oriented NoSQL database, a design that helps validate the functionality and scalability of the solution simultaneously. This paper also describes various phases of planning, development, and testing of the OMS solution that was executed for a large retailer in Europe to test an extremely high online sales scenario. An out-of-the-box configuration of the OMS with the feature support for database sharding was used to drive scalability. The exercise was a success, and it is the world's largest IBM Sterling Order Management benchmark in terms of sales order volume, to date.

Categories and Subject Descriptors
C.4 [**Computer Systems Organization**]: Performance of Systems – *Design studies, Measurement techniques, Modeling techniques, Performance attributes.*

General Terms
Management, Measurement, Performance, Verification

Keywords
Load Testing, Big data, Test Harness, Rapid Prototyping, Test Automation Tool, Document Oriented Storage, Order Management

1. INTRODUCTION
Retailers across the world increasingly find that customers are moving to multiple channels due to of the convenience they offer. Customers want the benefit of being able to shop from anywhere at trusted brands and get the best price along with flexible delivery options offered by retailers. This distinct benefit to consumers has resulted in a robust growth of online business in recent years and still continues. Large retailers who operate in verticals like groceries, general merchandise, and clothing would look at a single order fulfillment system, to handle multi selling channels. Such order fulfillment systems should have the flexibility to support multiple fulfillment scenarios and at the same time handle the volume of orders received by large retailers.

IBM Sterling Order Management System (OMS) [2] has been successfully used by many of the biggest retailers across the world. IBM was approached by a large retailer to confirm the feasibility of replacing their legacy order fulfillment system with OMS. The vision was to build a single order fulfillment platform to support all the online business across all the channels. The IBM team recommended using the database sharding feature because of the extreme volume requirements.

The performance requirements for this platform exceeded any published OMS benchmarks, and so the team decided to do a volume Proof of Concept (POC) combining solution and performance test. The objective of the volume POC was to confirm that an out-of-the-box OMS that employed database sharding would scale to meet the retailer's peak hour volume requirements. However, when the discussions started on the performance test framework, the existing tools and methodology seemed inadequate to generate the requisite volumes for the test.

1.1 The Product Solution
OMS is a J2EE application with a relational database back-end. All the asynchronous interfaces are based on messaging systems through which OMS interacts with other systems. The synchronous calls are typically Web Services that can be directly invoked through HTTP and HTTPS protocols. A typical solution consists of product code with customer-specific configurations and customized code.

The retailer's online business had a performance target of approximately 4 million order lines per hour with an average of 120 lines per order. The primary focus of the POC was the sales impact since such volume was never tested in any previous benchmarks. Therefore, this new benchmark should be for a solution configured over out-of-the-box OMS modeled for the retailer's fulfillment process. The retailer also required the POC to be done on a non-IBM hardware stack, operating system and database software (for which there are no published benchmarks available from IBM).

This POC should validate OMS' scalability on the technical and application capabilities to anticipate future growth of the retailer. Additionally, the POC should be carried out by enabling the sharding feature spreading the transaction load to four database servers. The business processes encompassed testing the fulfillment of orders and payment settlement and each state change involved integration with retailer's systems. The POC did

LT'15, February 01 2015, Austin, TX, USA
Copyright 2015 ACM 978-1-4503-3337-5/15/02…$15.00
http://dx.doi.org/10.1145/2693182.2693185

not need any inventory tracking. The item catalog was maintained outside of the OMS as per the retailer's business requirements.

1.2 The Big Performance Question

OMS is an Online transaction processing (OLTP) application with majority of the transactions related to order or inventory processing. The basic unit of processing in retail is an order line. Each OMS transaction is a composite transaction to process a single order line, a key differentiator with other transaction processing systems like banking or stock exchange. Such a transcation could involve several state changes, complex operations such as order scheduling and inventory processing and multiple database operations within order, inventory, statistics and audit tables.The volume target set by the retailer for the order flow was 4 million order lines per hour, which is about 20 times higher than the average peak volumes observed in the industry. The number of lines per order was 120 (a relatively larger number than the industry average), and this meant dealing with a large XML file for each order. An order XML would be around 50 Kb. Thus, to achieve the target, OMS required processing about 1200 order lines per second. Based on the initial sizing exercise, one test that runs for 3 hours would generate about 500 GB of data in the database.

The testing tool should be capable of generating test input files, equal to or more than the POC volume to provide adequate test coverage. Tools that typically drive performance tests for OMS are IBM's Rational Performance Tester [3] and HP's LoadRunner [4]. Open-source tools such as Apache's JMeter [5] and Grinder [6] also are widely used. Messaging services and Web Services push in XML input messages into the OMS system. The inbound data for OMS tests are primarily XML files that are generated by the load generator application and are consumed by an appropriate transaction in OMS. Create and transfer of a significant number of large input XML files posed a unique problem to conduct a performance test.

High cost of commercial performance tools has led many researchers to seek a fitting alternative. The paper by Chen et al., 'Yet Another Performance Testing Framework' [6] is an example of such research. Conventional tools would require powerful test servers and they provide limited flexibility. In their paper 'Design and Implementation of Cloud-based Performance Testing System for Web Services' [8], Zhang et al. describes a cloud-based approach to performance testing of web services. The tool provides a front-end to produce test cases that are executed in the cloud and helps simulate concurrent user access.

Because of the strict budget and time constraints for this volume POC, a conventional tool was not viable. Another feasible way was to create XML messages in advance, store and feed into the test system using scripts. The paper 'CaPTIF: Comprehensive Performance TestIng Framework' [9] by Mayer et al. present a similar approach. The paper states that 'the framework allows for the definition of well-defined test inputs and the subsequent scheduling and execution of structured tests.' Though the paper suggests ways of test case generation, the creation and storage of test data is a crucial aspect of the framework. According to the paper, 'CaPTIF stores the details on all modules and instances, test configurations, test cases, test inputs, and test results for all test runs in a relational database backend'. Using RDBMS as the back-end, the CaPTIF team stored the entire test input as a database blob for efficiency reasons. It is clear from the paper that this was done to eliminate significant overheads due to database joins when the test input was accessed. The paper continues to explain that the test input was stored as JavaScript Object Notation (JSON) [9] format to make it independent of the underlying database schema.

Creation and storage of XML messages prior to the test poses two issues. First, modification of a large XML file causes significant disk and Java Virtual Machine (JVM) overheads. Second, the tester may want to modify the order flow or create alternate flows before or during the test, to ensure test coverage and accommodate additional scenarios (by editing the XML messages). There is no easy and quick way (like SQL in a relational database) to locate an XML file during the test run containing specific data and modify. Using a fast drive, e.g., SSD for handling the XML files may not be feasible due to the cost.

Volume POC required creation of stubs to simulate systems around OMS for a normal order flow. The messages also had to be generated based on actual customer behavior and maintain the randomness of orders and several other fields in OMS. Another problem was failures during a performance test because of functional or system level issues. Usually, such failures require cleanup of the participating systems before the start of a new test and could consume significant time and effort. To restart a test from the exact point of failure is always difficult as the test framework would also require a reset.

2. BIG DATA-BASED TEST FRAMEWORK

In short, a test framework was needed that would store messages for several scenarios and quickly modify XML messages during the run. The framework was expected to have a distributed storage and should generate the peak-load without becoming a performance bottleneck. It would also need to be installed and scripted easily to meet the tight schedule set by the retailer.

2.1 Choice of MongoDB

MongoDB [10] is an open-source and a leading NoSQL database. It supports document-oriented storage and document based-querying, has MapReduce for aggregation and data processing. MongoDB can be easily installed, set up quickly and can be scripted using JavaScript. It allows to query data dynamically and provides a query syntax that feels very similar to SQL. Other popular NoSQL databases like Cassandra and HBase process full-scale large data, while MongoDB fits well for files of a certain size range, quickly and schema-free, that precisely matches the POC requirements.

MongoDB is efficient in handling large files and real-time data query. In their paper on 'A Real-Time Log Analyzer Based on MongoDB' [11] Qi Lv et al. describes the features of MongoDB for real-time applications. The article states that 'MongoDB provides well query performance, aggregation frameworks, and distributed architecture that is suitable for real-time data query and massive log analysis'. The paper continues to observe 'Our experimental results show that HBase performs best balanced in all operations, while MongoDB provides less than 10ms query speed in some operations that is most suitable for real-time applications.' In their paper 'MongoDB vs Oracle-Database Comparison' [12] Boicea et al. concludes that MongoDB is a more rapid database management system, a simple database that responds very fast.

MongoDB could store large documents, of any type in JSON format. Document storage of such proportion was ideal as the tests primarily deal with XML files. Moreover, it was found that the generation and manipulation of input and output XMLs were up to ten times faster than conventional methods (when tested

internally using HP LoadRunner and Grinder). Since XMLs are the drivers for the tests, it follows that the test logic can be built-in the stored content, rather than in scripts thereby reducing the requirement of scripting. Any of the quick scripting tools like Perl or JavaScript or even a programming language like Java could be used for data and content handling. Scripts can be written using the mongo shell in JavaScript that can manipulate data in MongoDB or perform administrative operations.

Usually, the test framework should also include a mechanism to mimic any synchronous or asynchronous interfaces that might be necessary. Such mimicking is achieved through the use of stubbed scripts, and these stubs are logic-driven and not content driven. The ease by which MongoDB allows document update, gives the tester real-time control over stubs (response data, response time, and data format), an edge over conventional static-data stubs.A response document in real-time can be generated and sent back to the system under test (SUT) with minimal system resources and scripting effort, when MongoDB is used.

A MongoDB collection is roughly the equivalent of tables in a relational database. MongoDB can maintain several test data patterns in many collections through replication and minor modification of the content. Multiple iterations of tests can thus be planned in advance and executed within short intervals. Comparing MongoDB collections help analyze patterns across tests and thereby verify the test accuracy. Response times, errors and other statistical data can be stored in real-time and analyzed using standard big data analytical tools.

The output XML can also be stored and compared with the expected results for functional validation of the OMS transactions. Storage and live analysis of test results give better control over the performance tests and helps detect errors in advance. If necessary these tests can be re-run with minimal time and effort using the MongoDB framework. Appropriate flags can be set in additional columns for each MongoDB collection that can denote the current position of the test. Scripts are to be written to check these flags when a re-run is initiated.

The document storage capability of MongoDB can be further utilized for storage of performance outputs like log files, heap, thread dumps and statistical data. These files can be analyzed side-by-side with test output data and other test results. Since data is now stored and archived for various load scenarios, reusing them in other OMS implementations is also possible with minimal effort.

2.2 The Test Strategy

During the discovery phase of the POC, details on hardware, network, interfaces, and business process were gathered. XML templates were collected during this period and deployed in two parts, one in the OMS system and the other in MongoDB. OMS components were configured to have all the standard functionality, configuration, web services, data loading, and item catalog.The strategy was to use MongoDB as the substitute for all external interfaces, custom functionality, load generation and interface responses through stubs. This substitution required XML data transformation and data modification.

A few input XML messages were created in the MongoDB collections (load generator), and the scripts were run to initiate a data flow. This way the solution prototype was functionally validated. The functional test harness evolved into the performance harness, by increasing the volume, with no additional scripting and testing effort. Thus, the MongoDB component operated as a functional, integration and performance harness that could be used to drive all types of tests, a complete Test Harness.

All functional and performance tests were run by the same team in a departure from established norms. The volume of messages is the only difference between the functional and performance tests in this framework. The functional test analysis was done with the help of solution team involved in prototyping. The test framework was developed side-by-side with the solution prototype, and the test team could validate each step through continuous testing, thereby reducing the cost of defects.

2.3 POC Hardware Configuration

The POC used OMS version 9.2.1, an out-of-the-box configuration and with application audits enabled. The relational database was Oracle Enterprise 11.2.0.3, application server Apache JBoss EAP v5.1 and middleware TIBCO 6.1. The load generator machine, a Red Hat Enterprise Linux (RHEL) box, was installed with MongoDB 2.4.9 and required software. Table 2 shows the configuration of the load generator.

Table 1. Load generator machine

Component	Details (for each LPAR)
One load generator LPAR (MongoDB and Perl /Grinder scripts)	E5-2637V2 @3.5GHz, 8 CPU Cores 24 GB RAM per server
Total of 1 Server	

Table 3 describes the hardware specifications for the OMS solution implementation. Application tiers, middleware, and the database tier was designed after a detailed sizing exercise. Note the total size of the load generation to the actual solution infrastructure for the application under test; clearly the cost of test server is a bare minimum.

Table 2. Solutioninfrastructure

Component	Details (for each LPAR)
Five database LPARs	E5-2637V2 @3.5GHz, 8 CPU Cores 64 GB RAM per server
Eight batch processing LPARs	E5-2637V2 @3.5GHz, 20 CPU Core 256 GB per server
One application LPAR	E5-2637V2 @3.5GHz, 8 CPU Core 64 GB RAM per server
One message queue TIBCO LPAR	E5-2637V2 @3.5GHz, 8 CPU Core 64 GB RAM per server TIBCO EMS version 6.1
Total of 15 Servers	

2.4 Test Preparation and Execution

Test execution started with the preparation of test data. This activity was to create multiple MongoDB collections and create XMLs corresponding to a typical sales order transaction, for e.g. create an order, modify or cancel an order. Once an XML template document is available, duplicating it and updating the columns for random attributes was relatively easy with MongoDB's powerful features. The input data for a full round of functional test was created, followed by making multiple copies for distributed storage. Various data profiles for several test iterations were created by making copies of MongoDB collections. Once the test data is ready, scripts were executed to put XML documents from MongoDB to OMS and receive response. For asynchronous interfaces, Perl scripts handled the

transfer and mashup of messages. Grinder was used to emulate web service calls.

Perl and Grinder scripts were used to read from MongoDB and write into the middleware or as POST messages. The output messages are also read by Perl and written back to MongoDB, thus recording the result of each of the transaction. The reads and writes were done real-time and it was found to be multiple times faster than the OMS system itself. For the integration stubs, transformation scripts that would modify XMLs for the required parameters would write into MongoDB. Each appropriate message is then fed back into OMS, thereby emulating live-stubs.

After each test, the collections are marked and archived for later analysis. As a last step in the execution, the log files, statistical data, error tables and other essential tables are imported into MongoDB for analysis. Figure 1 shows the MongoDB performance harness illustrated.

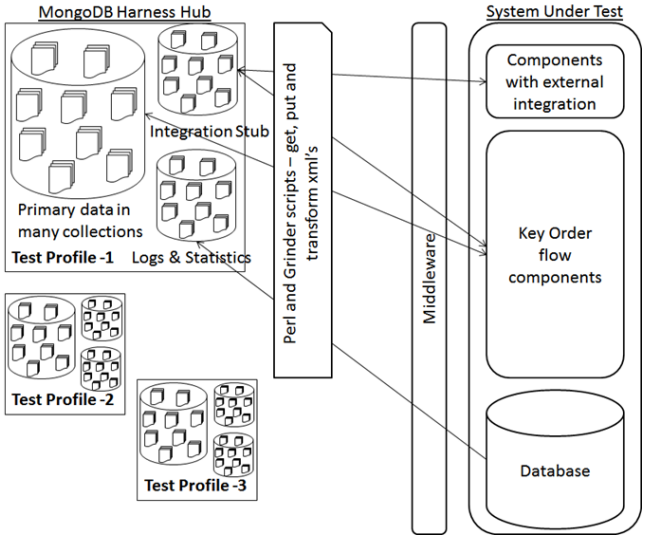

Figure 1. Test execution setup

2.4.1 Test Results

See Table 4 for the primary outcomes of the POC when the system reached a steady state. A typical test would last for 2 to 3 hours, and the results are taken for analysis. Multiple iterations of the same data profile are done to arrive at an average performance figure.

Table 3. POC Test Results

Component Name	Test Results (Throughput/hour)
Order creation	7.1 Million Order lines
Order amendment	464k Order line changes
Order cancelation	420k Order line cancelations
Authorization	5.6 Million Order lines
Delivery updates	5.56 Million Order lines
Settlement	5.54 Million Order lines
Web Service calls	298k Invocations per hour Average response time of 410 ms

The results of the POC also indicates the success of the MongoDB test framework. The framework could generate message volumes at even higher rates, but the tests were stopped when the OMS reached the POC goals. The average size of an XML file was about 35 to 50 Kb, and each MongoDB database operations (read, insert and update) took between 10 to15 milliseconds to complete. Each XML was modified for 5 to 8 fields and additional inserts to MongoDB columns were done for each.

MongoDB test harness helped solve the particular challenges that the volume performance test introduced. The ease of scripting, low-cost maintenance, high performance and minimum hardware requirements were the key wins of this approach. As the test logic resides in content, functional and performance tests can also be provided as a cloud-based service, although network security issues need resolution.

3. ACKNOWLEDGMENTS

Thanks to IBM RSC Bangalore team members Kamala Nagarajan, Nagalakshmi Vellaichamy and Anusha Dasari for their support in the POC activities, MongoDB data modeling and scripting.

4. REFERENCES

[1] Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I., & Williams, C. (2008). Data-Driven Testing. In Model-Driven Testing (pp. 87-95). Springer Berlin Heidelberg.

[2] "Sterling Order Management", DOI= http://www-03.ibm.com/software/products/en/order-management

[3] IBM: Rational Performance Tester. http://www-01.ibm.com/software/awdtools/tester/performance

[4] Hewlett Packard: HP LoadRunner. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/index.html

[5] Apache Software Foundation: Apache JMeter. http://jmeter.apache.org/

[6] Aston, P.: The Grinder, a Java Load Testing Framework. http://grinder.sourceforge.net/

[7] Chen, S., Moreland, D., Nepal, S., Zic, J.: Yet Another Performance Testing

[8] Mayer, D. A., Steele, O., Wetzel, S., & Meyer, U. (2012). CaPTIF: Comprehensive Performance TestIng Framework. In Testing Software and Systems (pp. 55-70). Springer Berlin Heidelberg.

[9] Chodorow, Kristina. MongoDB: the definitive guide. " O'Reilly Media, Inc.", 2013.

[10] Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational) (July 2006)

[11] Boicea, A., Radulescu, F., & Agapin, L. I. (2012, September). MongoDB vs Oracle-Database Comparison. In EIDWT (pp. 330-335).

[12] Lv, Q., &Xie, W. (2014, August). A Real-Time Log Analyzer Based on MongoDB. In Applied Mechanics and Materials (Vol. 571, pp. 497-501).