Performance Analysis of Lucene Index on HBase Environment

Anand Hegde & Prema Shraff
aghegde@indiana.edu & pshraff@indiana.edu

School of Informatics and Computing
Indiana University, Bloomington
### Table of Contents

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Topic</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Performance Analysis</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Details of our analysis</td>
<td>5</td>
</tr>
<tr>
<td>3.2</td>
<td>Results of our Performance Analysis</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Introduction to Survey</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Importance of performance test</td>
<td>9</td>
</tr>
<tr>
<td>4.2</td>
<td>Apache JMeter</td>
<td>9</td>
</tr>
<tr>
<td>4.3</td>
<td>LoadRunner</td>
<td>10</td>
</tr>
<tr>
<td>4.4</td>
<td>Grinder</td>
<td>11</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Future Work</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>Acknowledgements</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>References</td>
<td>15</td>
</tr>
</tbody>
</table>
1. Abstract:
The following report consists of combination of performance analysis of Lucene Index on HBase in HPC environment and a survey of various load testing frameworks. The results of the performance analysis and the survey are discussed in the following sections.

Data intensive computing has been an area of focus for many years and various technologies have been developed in order to store large amounts of data. Development of HBase system is a huge step in that direction. HBase system is basically modeled on Google’s BigTable. It supports reliable storage of data and efficient access to the stored data. However, various researchers have observed that HBase does not provide a good searching mechanism based on the columns.

2. Introduction:
A recent paper (Experimenting Lucene Index on HBase in an HPC Environment Xiaoming Gao, Vaibhav Nachankar & Judy Qiu) has addressed the issue of searching HBase using Lucene indices. We provide a brief overview of their implementation of Lucene Index on Hbase as a background for our study of various load testing frameworks.

HBase is basically Hadoop database. The diagram below explains how data is stored in HBase.

| Column families: BasicInfo, ClassGrades |
| Qualifiers: Name, Office, Database, Independent Study |
| Row keys: aaa@indiana.edu, bbb@indiana.edu |
| Version timestamps: t0, t1, t2, t3, t4, t5, t6 |

Table 1: HBase table architecture

Tables in HBase are split into various regions and are served by many region servers to access the data present in those tables. This kind of architecture ensures reliable data storage efficient access to huge amounts of data. The general flow is that a new client contacts the Zookeeper quorum (a separate cluster of Zookeeper nodes) first to find a particular row key. It does so by retrieving the server name (i.e. host name) that hosts the -ROOT- region from Zookeeper.

With that information it can query that server to get the server that hosts the .META. table. Both of these two details are cached and only looked up once. Lastly it can query the .META. server and retrieve the server that has the row the client is looking for.

Once it has been told where the row resides, i.e. in what region, it caches this information as well and contacts the HRegionServer hosting that region directly. So over time the client has a pretty complete picture of where to get rows from without needing to query the .META. server again.
Next the *HRegionServer* opens the region it creates a corresponding *HRegion* object. When the *HRegion* is "opened" it sets up a Store instance for each *HColumnFamily* for every table as defined by the user beforehand. Each of the Store instances can in turn have one or more StoreFile instances, which are lightweight wrappers around the actual storage file called *HFile*. A *HRegion* also has a *MemStore* and a *HLog* instance. We will now have a look at how they work together but also where there are exceptions to the rule.

The following diagram provides an overview of the HBase architecture. The following diagram describes the relation between ZooKeeper cluster, region servers and also data nodes.

![HBase architecture](image)

**Figure 2: HBase architecture**

The authors of the paper have made use of inverted indices using Apache Lucene which is a library written in Java for building inverted indices. The authors have made use of MapReduce programs for building indices.

In order to perform analysis of their implementation it is necessary for us to study various testing frameworks. There are many testing frameworks available in the market both open source and commercial products. Following are the testing frameworks we have studied and compared.
3. Performance Analysis:
As mentioned earlier, we have analyzed the performance of the Lucene index implementation. We performed our experiment on the Alamo cluster of FutureGrid. In order to understand our conclusions it is very important to know the configurations of the Alamo cluster on FutureGrid. Following are the characteristics of Alamo.

Following are the system details of Alamo on FutureGrid:

- System type: Dell PowerEdge
- No. of CPUs: 192
- No. of cores: 768
- Total RAM: 1152 GB
- Storage: 30 TB

We have executed the program both sequentially and in parallel across various nodes to conduct our analysis. In order to get accurate results, we took average of 5 readings for every configuration. Configurations were changed based on the size of the data and the number of nodes.

3.1 Details of our analysis:
In order to test the program of Lucene index on Alamo cluster, it is necessary to have 5 mandatory nodes. They are as follows:

- 3 ZooKeeper nodes
- 1 HDFS Master node
- 1 HBase Master node

Keeping this in mind we have tested the program by changing the number of data nodes depending on the size of the data being tested. Following are the steps we performed in order to complete our analysis.

- Configured Hadoop and HBase on Alamo cluster
- Added scripts to execute the program sequentially
- Modified the existing scripts to record the time of execution
- Modified the scripts to record the size of each table on HBase

We executed the program on multiple nodes with data nodes ranging from 6 to 8. We also tested the program for different sizes of data ranging from 50 MB to 30 MB. We can break down our performance analysis into three major parts. They are as follows:

- Sequential execution across same number of nodes for different data sizes.
- Sequential execution across different number of data nodes for same data size.
- Parallel execution across same number of nodes for different data sizes.
Following is the snapshot of the data in our Meta_Table which contains data about other data (i.e. data about BookIndexTable, BookTextTable, BookImageTable etc).

![Figure 3: Snapshot of the Meta Table](image)

Figure 3 explains the Meta Table which we used to map our data from different tables. Following is the description of the above diagram.

- A: This is the Row key for a particular entry
- B: This is the name of the qualifier
- C: This is the name of the Column family
- D: This the field contains the value

With the help of this data we could vary the size of the data require for testing since data is mapped from the meta table to the actual tables.

In order to vary the number of computational nodes, we made changes to the PBS script. Also, in order to record the size of the table we added an extra script. Also we made changes to record the time by editing the PBS script.
3.2 Results of our Performance Analysis:

3.2.1 In the first experiment, we executed the program sequentially across same number of computational nodes for different data sizes. Following is the graph we obtained after execution.

![Sequential execution](image)

**Figure 4: Number of nodes = 11**

The above figure explains the readings we obtained when the program was tested across 11 computational nodes (5 mandatory nodes + 6 data nodes). The results we obtained were expected. As the size of the data was increased the computation time increased linearly. Thus we conclude that the size of the data plays a very important role in sequential execution.

3.2.2 In the second experiment, we executed the program sequentially across different number of computational nodes for the same data size. We observed a rare phenomenon in this case. One would expect the time to decrease as the number of nodes were increased assuming that more number of nodes meant more computation power. We observed that was not the case.

In fact as the numbers of nodes were increased we observed the time to increase as well (Figure 5). In this experiment we took a small data size of 50 MB and executed it across 11, 13, 15, 17 and 19 nodes. The computation time increased because for such a small data size, the communication latency between the nodes in sequential execution becomes large. The time for communication between the nodes increases thus increasing the total computation time.
Following are the observations of the third experiment we conducted (Figure 6). Here we executed the program in parallel across multiple nodes.

### Figure 6: Size of the data = 50 MB
The above figure explains the readings we obtained when the program was tested across 13 computational nodes (5 mandatory nodes + 8 data nodes). The results we obtained were expected. As the size of the data was increased the computation time increased linearly. We observed that time for execution for a given data size is much less when executed in parallel as compared to sequential execution.

4. Introduction for Survey:
Performance testing is an important part of any distributed or web application testing plan. Including a performance estimate into the planning and development cycle of a project ensures that the application can hold high load. This gives an assurance that the program can meet the scalability requirements.

4.1 Importance of performance test:
It is important to identify the amount of load that a particular software can handle. It can help configure the system appropriately and avoid unexpected crashes. Every software has a concern whether it will be able to hold the request coming from hundreds and thousands of clients that is what amount of frequency can it handle. This type of test not only provides an absolute measure of system response time, but also targets the regressions on server and application code, examines if the response from the server matches the expected result, and helps to evaluate and compare middleware solutions from different vendors.

Hence before starting to implement a test framework for the Lucene index project we thought of surveying various existing project testing frameworks to select the best possible option.

1. Apache JMeter
2. LoadRunner
3. Grinder

4.2 Apache JMeter:
Apache JMeter is a Java desktop application from Apache. It is designed to do load testing of the functional behavior and measure the performance. It is purely a java application. It was originally build for testing web applications. Apache JMeter can be used to test performance on both static and dynamic resources. These dynamic resources include files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers and more.

It can be used to analyze overall server performance under simulated heavy load. The software features FTP and HTTP requests and extensible custom scripting features. Apache JMeter can be used to simulate a heavy load on a server, network or object to test its strength. It can also be used to analyze overall performance under different load types. For analysis of performance or for testing the server, script or object behavior under a heavy load we can use JMeter.

A typical JMeter test involves creating a loop and a thread group. The loop simulates sequential requests to the server with a preset delay. A thread group is designed to simulate a concurrent load. JMeter provides a user interface. It also exposes an API that allows you to run JMeter-based tests from a Java application. To create a load test in JMeter one can build a test plan, which is essentially a sequence of operations JMeter will execute. It generally consists of thread group, samplers and
listeners. Thread group is used to specify the number of running threads and a time for the initial ramp up. Each thread simulates a user and the time for ramp up specifies the time required to create all threads. The loop count defines the run time for a thread. The scheduler allows one to set the start and end of the run time. The samplers are configurable requests to the sever HTTP, FTP or LDAP requests. The listeners are the elements that are used to process request data. The results can be saved.

4.3 LoadRunner:
It is an automated performance testing product on a commercial ground. It can be used for examining system behavior and performance. While running a test by LoadRunner the system is loaded to find how it reacts and how it affects the performance. LoadRunner can emulate hundreds or thousands of concurrent users to put the application through the rigors of real life user loads, while collecting information from key infrastructure components like Web servers, database servers.

Load Runner framework:
The framework supports the creation of a series of business processes or business threads that are captured as virtual user scripts. LoadRunner enables the testing team to organize, author, and maintain a set of Virtual User Scripts and supporting software libraries in a shareable environment. These scripts were then be used to simulate several users (hundreds to thousands) accessing the application at once by using the LoadRunner Controller. LoadRunner provides a software development environment designed to meet the needs of experienced Test Automation Engineers. Two primary development languages are supported by LoadRunner JavaScript and C-script. This makes the development environment accessible to anyone with development experience.

Virtual User Scripts:
LoadRunner provides a simple record and playback mechanism that can be used to create Virtual User Scripts. These scripts become the baseline for developing a scenario or business thread that will be used to exercise the application under test. The Virtual User development environment allows the Test Automation Engineer to customize the initial scripts by defining transactions, defining rendezvous points, controlling playback behavior, supporting data driven scripts (parameterization), and full customization of the underlying code.

Controller:
Controller allows the Test Automation engineer to execute several Virtual User Scripts each script being executed by 1 to n virtual users. The Controller can use 1 to n stations to execute these scenarios. The Controller provides a full suite of basic monitoring screens that can be used to measure the performance of the application under test from several perspectives like throughput, response time and transaction failure rate. These results can be saved to a report that can be viewed at any time. Controller enables the Tester to control several aspects of the Performance test. These include the virtual user scripts, number of virtual users per script, ramp up time, ramp down time, ramp up/down of virtual users, execution time. Thus it allows a full customization of what, where, when, and how much testing will occur for that run. This is not an ad-hoc assembly of wizards. It is a well thought out solution that is designed to support performance and load testing.
Feedback on LoadRunner:
Thus LoadRunner is a powerful performance-testing tool able to deal with almost any architectural framework you may want to test. However LoadRunner is very dependent on the accuracy of the virtual user script. Most of the issues encountered during script execution can be due to changes in the environment or user setup. LoadRunner comes with the capacity to create standalone code and store them in sharable libraries. This allowed the automation engineers to create a common toolkit that could be accessed and maintained by the team. Load Runner does allow the Test Automation Engineer to use Quality Center as a common repository of scripts that can be accessed by all members of the team. However there is a very high price to license the software. Generating unlimited load is not permitted within LoadRunner.

4.4 Grinder:
Grinder is an open source framework providing a simple way for the developers to perform load testing. Grinder makes use of Jython scripts and these scripts are distributed over the network and run by agents. It has a simple framework in which there is a property file that defines load test that have to be run. Also the script contains details about the number of times and no of threads on which the script is to be run. A central application console starts the agents, collects the responses, and monitors the number of requests per second.

Grinder framework:
It contains a local console and local agent running workers. Thus there are four components agents, workers, console and TCP proxy. There will be a single agent on each load injector machine, which will start a configured number of worker processes. If the agents can connect to the console, they’ll wait for a signal to start before passing off a local `grinder.properties` file to the worker processes. The workers are the one that do the work. They actually execute the load test scripts. The `grinder.properties` file that is passed to the worker by the agent defines, among other things, the script that the worker will execute against the target, how many threads the worker will spawn, and how many times each one of those threads will execute the script.

The console sits on top of everything. It is the controller that pushes scenarios and test cases to the agents that run on load injection machines. It is a GUI that can be used to control the agents, and also displays the collected statistics that are reported back from the workers. Each agent is in charge of one or more workers that inject load into the system being tested.

The workers report the data back to the console. Then the console aggregates all the data that it receives from the workers and presents the output to the tester. The TCP proxy lies between the browser and the server. Its job is to echo whatever goes through it. It can be used to generate scripts after recording the activity of the browser. This is useful in the case where we need tests that simulate user interaction with a web application.

Grinder test scripts are written in Jython. Jython is a Java implementation of Python that is compiled to byte code and runs on a JVM and also allows you to access the Java libraries using the Python syntax.

**Feedback on Grinder:**
The Grinder remains a simple tool to set up and start testing web applications under heavy load. Running Windows Performance Monitor in parallel allows correlating the number of requests per second with underlying performance counters. Thus we would recommend *The Grinder* to developers who would like to ensure that their web applications respond well under stress before handing them to more experimented testers knowing the in-and-outs of more sophisticated testing tools.
4.5 Comparison:

Following table briefly explains the pros and cons of the three testing frameworks we studied.

<table>
<thead>
<tr>
<th>Sr No</th>
<th>Parameter for comparison</th>
<th>LoadRunner</th>
<th>Grinder</th>
<th>Apache JMeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Server monitoring</td>
<td>Strong for windows. Weak for Unix</td>
<td>Uses the wrapper based approach to sync the test data with the monitoring data.</td>
<td>Has no monitoring built in.</td>
</tr>
<tr>
<td>2.</td>
<td>Amount of load generated</td>
<td>Restricts the number of users. However generates extreme load per user and hence requires less hardware.</td>
<td>No of agents is restricted. Cannot effectively down load large files.</td>
<td>You can run as many agents as much as hardware support is available. Ability of agents falls sometimes due to memory issues.</td>
</tr>
<tr>
<td>3.</td>
<td>Can run in batch (non-interactive mode)</td>
<td>No. Multiple scenarios can’t be launched from the command line. Not the ideal test for running automated batch tests</td>
<td>Can run Single test from command line.</td>
<td>Yes. Can run batch jobs. UI required for distributed testing.</td>
</tr>
<tr>
<td>4.</td>
<td>Ease of use: Installation</td>
<td>Difficult and time consuming. Occupies large disk space.</td>
<td>Easy to install</td>
<td>Easy to install</td>
</tr>
<tr>
<td>5.</td>
<td>Running tests</td>
<td>Complex in execution.</td>
<td>Properties.config file needs to be configured. Agent and console need to be started.</td>
<td>Both distributed and local tests can be started form the UI.</td>
</tr>
<tr>
<td>6.</td>
<td>Setting up tests</td>
<td>Icon based scripts make it easy to run HTTP tests.</td>
<td>Scripts can be set up using jython.</td>
<td>Simple. Just a thread and a sampler need to be added.</td>
</tr>
<tr>
<td>7.</td>
<td>Generation of Results</td>
<td>Has an integrated analysis tool for generating dynamic graphs. However small number of performance metrics.</td>
<td>There are no graphs generated out of the box.</td>
<td>JMeter does not gather any server-side performance metrics. But it can generate a limited number of client-side graphs</td>
</tr>
<tr>
<td>8.</td>
<td>Tools for analysis</td>
<td>Powerful</td>
<td>No tools available.</td>
<td>Post analysis not very good</td>
</tr>
<tr>
<td>9.</td>
<td>Agent management</td>
<td>Each agent can run as a service or an application thus making management easy.</td>
<td>Properties file defining the amount of load needs to be manually deployed to all the agents.</td>
<td>Each agent is a server that connects to the controller in real time. It offers</td>
</tr>
</tbody>
</table>
### 5. Future Work:
Following are the lists of actions we think are relevant as part of the future work of our term project.

- We have performed our tests on the Alamo cluster. We plan to perform similar tests on the India cluster of FutureGrid.
- From our survey, we conclude that Grinder suits our needs and we plan to use Grinder to validate our testing scripts.
- We plan to perform the same tests for even large amount of data. We plan to scale the data size up to 100 GB.

### 6. Conclusions:

- Sequential execution on the Alamo cluster for the Lucene Index program takes more time compared to parallel execution of the same on HBase.
- Research indicates that HBase is not as robust as the BigTable yet.
- From the survey we conclude that Grinder is an open source framework providing a simple way for the developers to perform load testing.
- Grinder also provides good real time feedbacks.

### 7. Acknowledgements:
We would like to thank Professor Judy Qiu and Stephen for their support during the entire course. We would also like to thank Xiaoming Gao for his support and assistance. His experiment was the source of inspiration for us.
8. References:

- http://grinder.sourceforge.net/
- http://jmeter.apache.org/
- Experimenting Lucene index on HBase in an HPC environment