Building a full-text index

on a NoSQL Store

MASTER THESIS

THI THU HANG NGUYEN

August 2013

Supervisors:

Prof. Dr. Philippe CUDRÉ-MAUROUX
Dr. Gianluca DEMARTINI

UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

the eXascale Infolab
Department of Informatics
University of Fribourg
Switzerland

eXascale Infolab

Abstract

My interest in the research project 'Building a full-text index on a NoSQL Store' grows out of a master program's course on Advanced Database Systems delivered by Prof. Dr. Philippe CUDRÉ-MAUROUX, and the seminar 'NoSQL Databases : Cassandra' presented by Benoit PERROUD, Software engineer at Verisign, during the Spring semester 2012.

NoSQL databases are divided into four main categories:

- Wide Column Store, also known as Column Families
- Document Store
- Key Value, also known as Tuple Store
- Graph Databases

In other to implement a full-text index, Apache Cassandra is chosen, which belongs to the Column Families category.

The chapters of this research report will be structured as follows.

Chapter 1 presents the technologies involved in this research: the n-gram model, the inverted file index, Apache Cassandra, and Apache Solr which is an enterprise search engine. In Chapter 2, the environment setup will be described step by step as required by Apache Cassandra, and Apache Solr. Each environment will be tested against a small sample of search queries before the experimentation phases. In Chapter 3, a set of 250 search queries will be performed according to an experimentation plan on Apache Cassandra and Apache Solr. Performance data will be recorded during the experimentation and compared.

For brevity, the full product name Apache Cassandra is shortened to Cassandra, and Apache Solr to Solr.

Contents

List of Figures

List of Tables

Source Code Listings

1 Introduction

1.1 Objectives

Today, one of important issues in database-oriented applications is searching on big volume of data to satisfy user information needs. Large amount of available data raises other issues to be addressed, particularly, the performance of query processing and the reliability of database systems.

In research literature and in industry, different approaches to searching were published and marketed to cope with those challenges. The contribution of this master thesis is to implement a search solution which uses n-gram model to create inverted index in Apache Cassandra. The performance of this implementation will then be compared with Apache Solr search engine.

In Section 1.2, n-gram at the word level will be discussed. Section 1.3 presents an indexing technique called inverted file combined with n-gram model. Section 1.4 examines the mechanism of read and write operations and how data consistency is maintained. In Section 1.5, a brief overview of Apache Solr will be given.

1.2 N-gram at the word level

N-gram is defined as "*a sequence of variable characters that stands for a word or string of words in a corpus*" (Dictionary.com). More precisely, a corpus is extracted by sliding into sequences with a fixed-length (n) without linguistic meaning.

Depending on application context and natural language, we choose n-gram's parameters (characters, words, stems) in order to obtain the desired search performance. For example, each n-gram string consists of serial n characters for Asian languages such as Japanese, Korean, and Chinese. In Latin languages, English for example, n-gram strings are extracted based on words.

Let us consider the following text, "the weather is well today." Then, 3-grams generated as sequences of characters from this sentence are: "the", "he ", "e w", …, "day", "ay ", "y ". 3-grams based on words are: "# the weather", "the weather is", "weather is well", "is well today" and "well today #". In this case, # is a special character added to the first 3-gram and to the last one.

To resume, n-gram models can be imagined as placing a small window over a sentence or a text, through which only n words or n characters are visible at the same time. For this research, only words will be considered to build unigrams, bigrams, and trigrams. Unigram is the simplest n-gram model. In fact, a search query used unigram gives a large volume of results. It could be called irrelevant results, and n-gram starts to become interesting when n is two (a bigram) or greater.

In the next section, we will discuss about the inverted file indexing based on n-gram model, its problems that could occur, and how to eliminate those problems.

1.3 Inverted file indexing

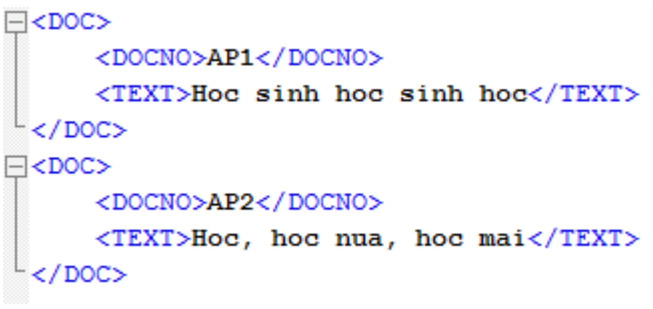
1.3.1 Indexing based on n-gram model

Indexes help speed up retrieval of data without scanning the entire dataset. We cannot predict the search strings that people will use to query a database. Each word in a document could be part of a potential search term. This suggests that indexing on words is a good strategy to speed up retrieval of relevant documents.

In an inverted file, each entry consists of a key, its frequency of occurrences and the associated document IDs. Each key can be atomic in unigram or complex in bigram, trigram. Keys are case insensitive in this research.

To illustrate the n-gram and to explain how an inverted file is created, let's assume a set of two vietnamese documents identified by `AP1`, and `AP2` as shown in Figure 1.1. The first text '`hoc sinh hoc sinh hoc`' means '`students study biology`', and the second one means '`Study ,study more ,study forever`'.

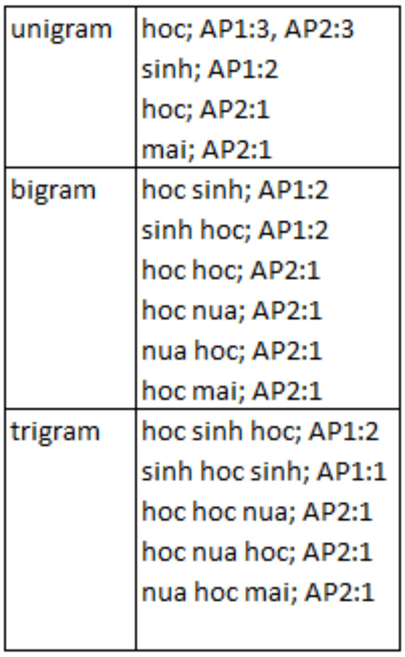
Figure 1.1
Two vietnamese documents to be indexed



```
<DOC>
    <DOCNO>AP1</DOCNO>
    <TEXT>Hoc sinh hoc sinh hoc</TEXT>
</DOC>
<DOC>
    <DOCNO>AP2</DOCNO>
    <TEXT>Hoc, hoc nua, hoc mai</TEXT>
</DOC>
```

From the text of `AP1`, trigrams based on words are '`hoc sinh hoc`', and '`sinh hoc sinh`'. A full inverted file including unigrams, bigrams, and trigrams is shown in Figure 1.2.

Figure 1.2
Inverted file from documents AP1, and AP2



unigram	hoc; AP1:3, AP2:3
	sinh; AP1:2
	hoc; AP2:1
	mai; AP2:1
bigram	hoc sinh; AP1:2
	sinh hoc; AP1:2
	hoc hoc; AP2:1
	hoc nua; AP2:1
	nua hoc; AP2:1
	hoc mai; AP2:1
trigram	hoc sinh hoc; AP1:2
	sinh hoc sinh; AP1:1
	hoc hoc nua; AP2:1
	hoc nua hoc; AP2:1
	nua hoc mai; AP2:1

As the length of texts, and the number of documents increase, the inverted file size is likely to become bigger. One of the problems with n-grams is that a search query could yield an excessive number of irrelevant matches. Indeed, some words are so common that almost every document contains them, such as “a”, “the”, “it”, …; in certain domains, e.g., information technology, a set of domain-related words frequently appears, such as ‘`computer`’, ‘`network`’, … It is useless to index them. They are called stop words. Irrelevant matches could be reduced, but not completely eliminated, by filling a list of stopwords as much as possible for a given domain.

Another issue is that indexing words having the same root in natural language, verbs in different tenses, nouns in singular and plural forms, could yield an excessively big inverted file. To deal with this challenge, stemming is used to to reduce the inverted file size, and to enhance thereby the performance of index.

For example, a stemming algorithm reduces:

- the words "fishing", "fished", "fish", and "fisher" to the root word, "fish";
- the words "argue", "argued", "argues", "arguing", and "argus" to the stem "argu";
- the words "argument" and "arguments" to the stem "argument".

1.3.2 Inverted index creation

The procedure to create an inverted index from a set of data on 28'000 scientific papers is pseudo-coded by the following algorithm:

```
For each document
    load its content
    remove stopwords from content
    stem related words
    create new document resulting from above operations
    create n-gram (unigram, bigram, trigram) from new document
    For each n-gram
        count frequency of occurrences
        create new entry (n-gram, docID, freq) in inverted file
    End For
End For
```

After processing all 28'000 documents, Cassandra contains an optimized inverted file in which stopwords have been removed and related words stemmed.

1.3.3 Query processing

The procedure to process a query against Cassandra database is presented by the following algorithm in pseudo code:

```
For each query
    remove stopwords from search string
    stem related words
    create new search string from above operations
    create n-gram (unigram, bigram, trigram) from new search string
    For each n-gram
        make a query on Cassandra database
    End For
    produce final query results using operator specified by user (AND,
OR)
End For
```

1.3.4 Ranking of results

A search query execution might result in zero, one, or many documents. The occurrences of the search string within each document found differ from one document to another. For this reason, the search engine normally ranks the documents in decreasing order of the occurrences in each document.

We will not discuss the details of the ranking mechanism, since this topic is out of our research scope.

1.4 Cassandra

1.4.1 Overview: Cassandra, a NoSQL approach

Cassandra is a nonrelational database management system designed to run on clusters with great flexibility. The design of nonrelational databases does not rely on relational schema. For the proponents of the NoSQL database approach, NoSQL systems are more efficient than relational database systems especially in "Big data" applications, scale much better and are more flexible to adjust to changes in data structure. Indeed, in NoSQL database systems such as Cassandra, new fields can be freely added to database records without having to redefine first the data structure (schema). Scalability means that, for example, if a high volume of data requires more processing power, then new machines can be added into the cluster without stopping operation.

NoSQL data models can be categorized into four groups: key-value, document, column-family and graph. Cassandra belongs to the column-family category. Column-family data model is discussed in (Sadalage, Fowler, 2013).

Database in Cassandra is structured as columns. The column is the basic unit of storage. Each column consists of a name-value pair where the name behaves as the key. Each name-value pair is a single column always stored with a timestamp. The timestamp is used for many purposes: data expiration, resolving write conflicts, dealing with stale data, etc.

We now study how Cassandra fulfills the three requirements stated in the CAP theorem (**C**onsistency, **A**vailability and **P**artitioning) (Hewitt, 2011; Datastax Apache Cassandra 1.0 Documentation; Datastax FAQ).

- *Consistency*: All database clients will read the same value for the same query, even given concurrent updates.

Cassandra has mechanism to synchronize up-to-date data on all replicas. Furthermore, tunable consistency for client request is possible at different consistency levels for both read and write depending on client requirements for response time versus data accuracy.

- *Availability*: All database clients will always be able to read and write data.

Cassandra is a distributed database management system designed for high availability and fault tolerance. There is no single point of failure in Cassandra clusters thanks to automatic data replication between nodes in the clusters, between physical server racks, between geographically dispersed data centers.

- *Partition Tolerance*: Partition is a communications break between two nodes which are both up. Partition tolerance means that the cluster continues to function even if there is a partition between nodes.

Cassandra always offers partition tolerance. In the sense of CAP theorem, a system with high partition tolerance and availability like Cassandra will give up some consistency in order to do it.

Cassandra has a query language that supports SQL-like commands, known as Cassandra Query Language (CQL). CQL has many more features for querying data but not all the features that SQL has, for example, joins and sub-queries. Basic queries on Cassandra include the GET, SET and DEL. The GET command is used to read data back from either a whole column family or a desired column from the column family. If we want to update data or create new ones, the SET command is used. With DEL command, we can delete either a column or the entire column family.

1.4.2 Cassandra data model

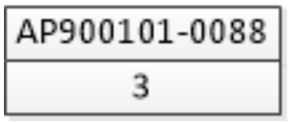
Let's define the main components of a data model in Cassandra: column, column family, super column, and super column family.

A Column is a tuple containing a name, a value and a timestamp represented by name:value:timestamp. The following example gives a column with name AP900101-0088, value 3, and timestamp 123456788.

```
{
name: AP900101-0088,
value: 3
timestamp: 123456788
}
```

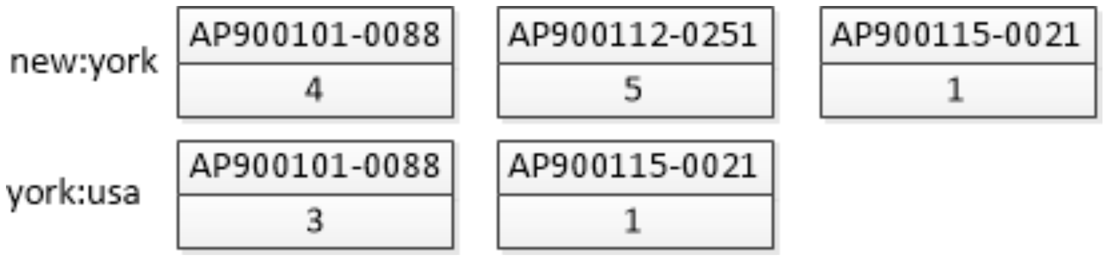
For simplicity, the couple name:value will be used to define a column in our discussion.

Figure 1.3
A column without timestamp



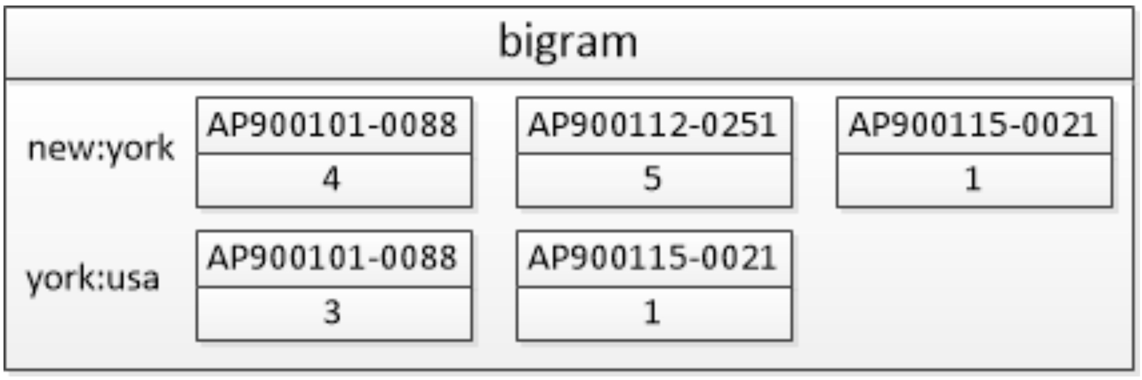
A Column Family is a couple name:value, where name is the column family name which is also called row key, and value is an array of columns as defined above. In Figure 1.4, the column family is named new:york, its value is an array containing three columns sorted by column name.

Figure 1.4
Two column families with row keys new:york and york:usa



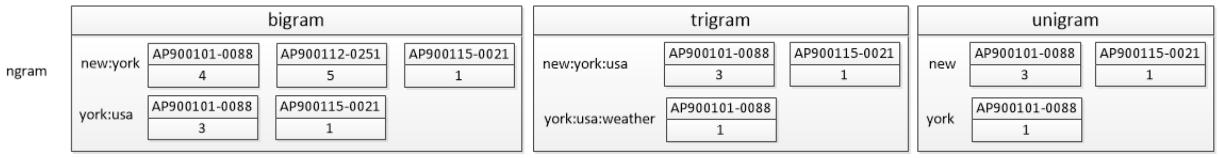
A Super Column is a couple name:value, where name is the super column name, and value is an array of column families as defined above. The array is sorted by column family name. In Figure 1.5, the super column is named bigram, its value is an array of two column families: new:york and york:usa.

Figure 1.5
A super column named bigram



A `Super Column Family` is a couple `name:value`, where `name` is the super column family name, and `value` is an array of super columns sorted by super column name. In Figure 1.6, the super column family name is `n-gram`, its value is an array containing three super columns: `bigram`, `trigram` and `unigram`.

Figure 1.6
A super column family named n-gram



1.4.3 Cassandra architecture

The architecture of Cassandra is made of three building blocks: `memtable`, `sstable`, and `commitlog`. These three components and their dependencies are best explained by examining the mechanism of a WRITE operation in an individual node.

a. commitlog and memtable

Let's assume that a query contains the following data to be processed: `column_family_name;column_name:column_value`, for example, `new:york;AP900101-0088:3`.

When the above data write request (`new:york;AP900101-0088:3`) is sent to a Cassandra node, two operations will take place (Figure 1.7).

- The first one is simple to understand: data are appended to the `commitlog`.

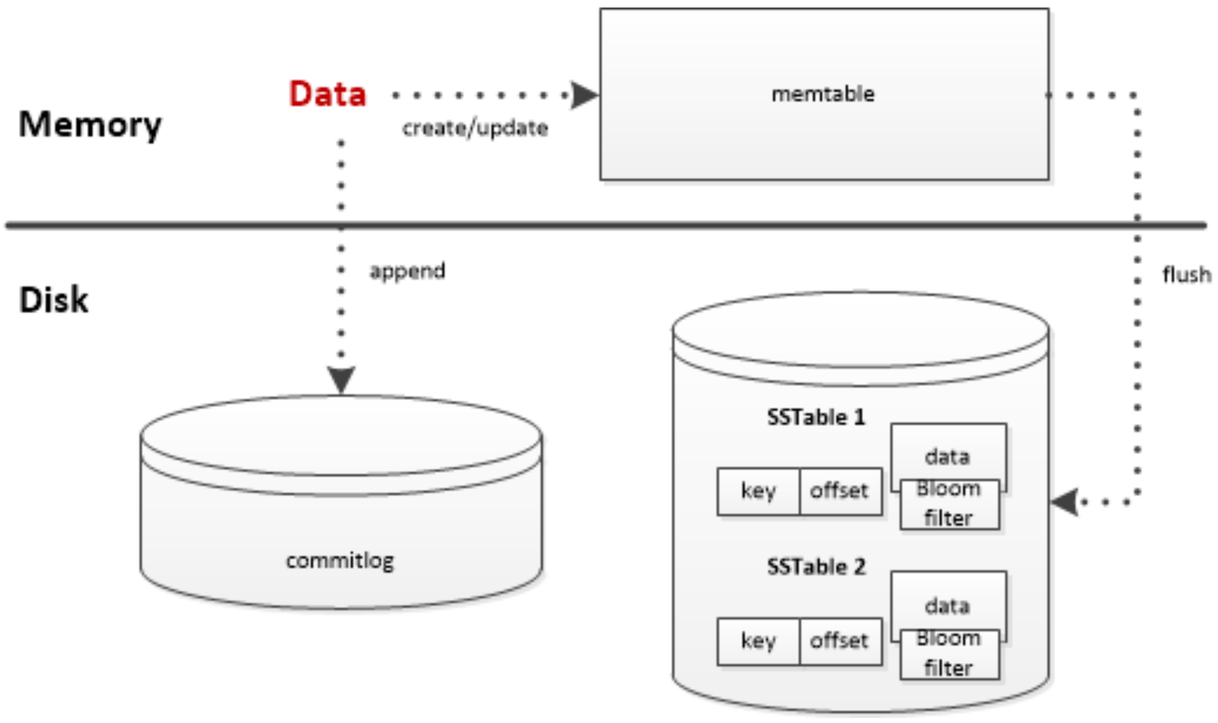
- The second operation affects a `memtable` whose contents are structured as column families identified by column family name, also known as row key. Two cases must be distinguished to process the data write request:

- **Case 1**: The data contains a row key (`new:york`) already existing in `memtable`.

- If the column name (`AP900101-0088`) already exists in the column family identified by `new:york`, then that column will be updated with the new value `3` within column family.
- If `AP900101-0088` does not exist, a new column `AP900101-0088:3` will be created in the `new:york` column family.

▪ **Case 2**: `memtable` does not contain any column family whose row key is `new:york`. A new column family will be created with column family name `new:york` and column family value `AP900101-0088:3` (one-element array).

Figure 1.7
Processing of data write request



`commitlog` acts as a crash recovery log for data. Once both `commitlog` and `memtable` are written, the processing of data write request is successfully completed.

b. sstable

When `memtable` is full, its contents will be flushed into `sstable` which is located on disk.

An `sstable`, which stands for Sorted String **Table**, includes three structures as shown in Figure 1.8 the data stored in file `Data.db`, a row index in `Index.db` and a bloom filter in `Filter.db`. Each `sstable` owns two additional files, `CompressionInfo.db`, and `Statistics.db` to store information about data compression, and statistics about row size, column count, etc., respectively.

- Data in `sstable` are maintained per column family as in `memtable`. Since `sstable` could not be changed, the next flushed data will be written to a new `sstable`.

- Bloom filter is used to perform key lookup efficiently.

Cassandra has an interesting background process called `compaction` whose purpose is to prevent the degradation of read performance. Indeed, each data flush requires a new `sstable`. Hence, a given column family could be flushed into several `sstables`. Thus, for example, retrieving a column value with a given row key and column name might require several reads. As the number of `sstable` increases, the read speed will deteriorate. To prevent it, the `compaction` process merges `sstables` into a new `sstable`, and recreates new corresponding indexes. Row keys will now be recorded at one and only one place.

1.4.4 Write in Cassandra

In this section, the write operation in Cassandra will be analyzed first in an individual node, then in the entire cluster with a single data center.

a. In an individual node

We will explain how a Cassandra node deals with the following three data write requests. Note that the third request has the same row key as the first one, `new:york`.

```
write (new:york, AP900101-0088:3)
write (york:usa, AP900101-0088:3 AP900115-0021:1)
write (new:york, AP900101-0088:4 AP900112-0251:5 AP900115-0021:1)
```

The first request causes `memtable` and `commitlog` to be filled as follows:

```
memtable      new:york, AP900101-0088:3

commitlog     new:york, AP900101-0088:3
```

Executing the second request updates `memtable` and `commitlog` as follows:

```
memtable      new:york, AP900101-0088:3
              york:usa, AP900101-0088:3 AP900115-0021:1

commitlog     new:york, AP900101-0088:3
              york:usa, AP900101-0088:3 AP900115-0021:1
```

In the third request,

- the column name `AP900101-0088` in `new:york` column family already exists in `memtable`. Therefore, only the corresponding column value needs to be updated from 3 to 4.

- The two columns `AP900112-0251:5` and `AP900115-0021:1` are new.

`memtable` and `commitlog` are updated as follows after processing the third request:

```
memtable       new:york, AP900101-0088:4 AP900112-0251:5 AP900115-0021:1
               york:usa, AP900101-0088:3 AP900115-0021:1

commitlog      new:york, AP900101-0088:3
               york:usa, AP900101-0088:3 AP900115-0021:1
               new:york, AP900101-0088:4 AP900112-0251:5 AP900115-0021:1
```

Let's assume that `memtable` is now full, i.e., reaches a predefined threshold. At that moment, its contents are flushed into a new `sstable` as shown belows along with the creation of associated row index and bloom filter.

```
sstable        new:york, AP900101-0088:4 AP900112-0251:5 AP900115-0021:1
               york:usa AP900101-0088:3 AP900115-0021:1
```

Cassandra maintains a log to record all events which occur with write operations as shown in the excerpt below:

```
INFO 00:57:41,234 flushing high-traffic column family
CFS(Keyspace='helen', ColumnFamily='ngram') (estimated 53706918 bytes)
 INFO 00:57:41,234 Enqueuing flush of Memtable-
ngram@965617323(1838366/53706918 serialized/live bytes, 63389 ops)
 INFO 00:57:41,235 Writing Memtable-ngram@965617323(1838366/53706918
serialized/live bytes, 63389 ops)
 INFO 00:57:41,719 Completed flushing
/var/lib/cassandra/data/helen/ngram/helen-ngram-hf-316-Data.db (1827443
bytes) for commitlog position ReplayPosition(segmentId=1363086294640,
position=19491818)
```

b. In the cluster

Let’s consider a cluster of 8 nodes numbered from 1 to 8 with replicator factor N = 3.

A data write request (`new:york, AP900101-0088:3`) is sent by a client to a node in the cluster, say, node 7. Node 7 will send that data write request to each of N = 3 replica nodes (say, nodes 1, 3 and 4 in Figure 1.7), then wait for their write success or failure responses.

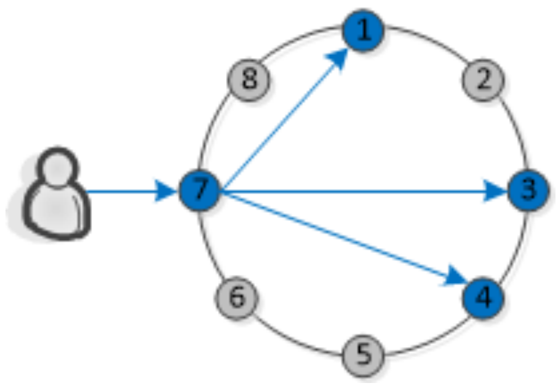
Consistency level sets the number of successful writes before sending an acknowledgement to the client. The possible values are: ANY, ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL. For instance, if the consistency level is preset to:

- ONE, at least one replica node must write successfully, i.e., both `commitlog` and `memtable` of that node are written.
- ALL, all three replica nodes must write successfully.

One node, say, node 1, dies and cannot receive the data write request from node 7. When node 1 is back to life, Cassandra uses a `hinted handoff` technique to make available the data write request to node 1 which will then process it.

Even if `hinted handoff` does not work, the stale data can be fixed by the `read repair` or `Anti-Entropy Node Repair` features of Cassandra to maintain data consistency.

Figure 1.8
Write on cluster



1.4.5 Read in Cassandra

In this section, the read operation in Cassandra will be presented first in an individual node, then in the entire cluster with a single data center.

a. In an individual node

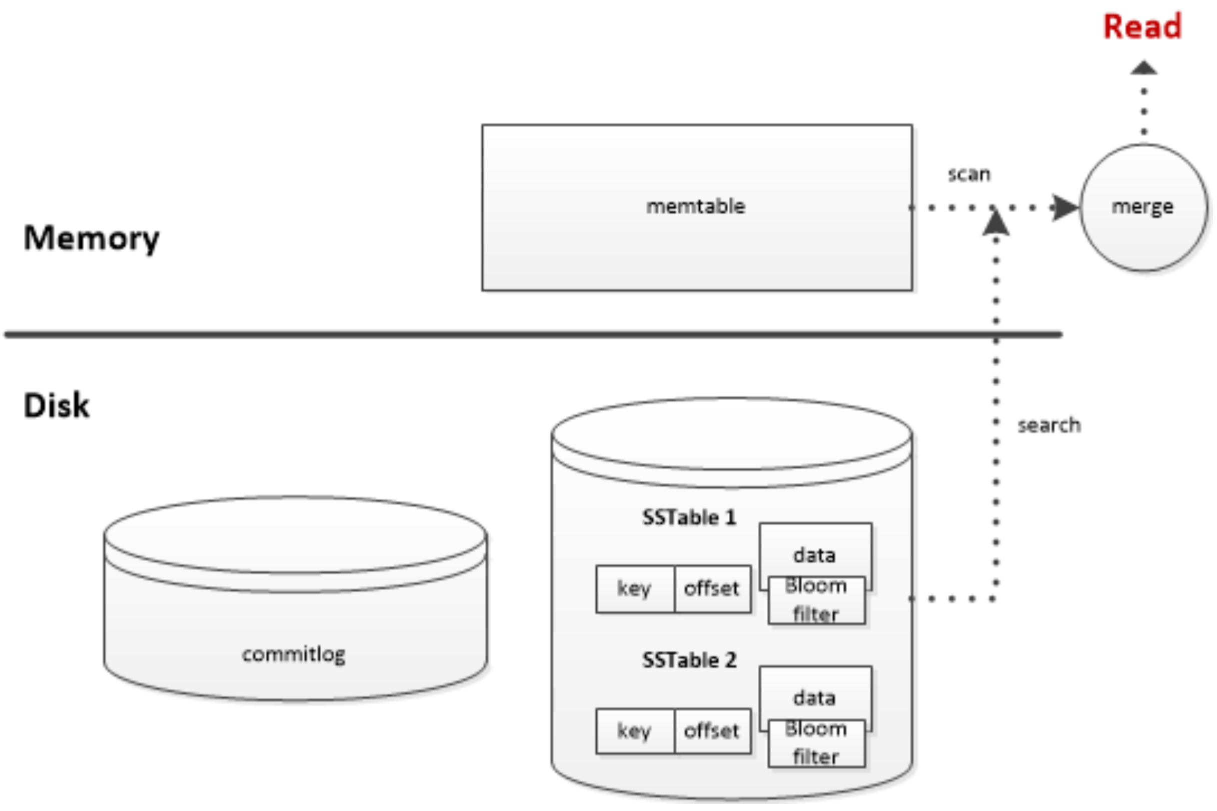
Let's assume that a query consists to retrieve a whole column family whose row key is, for example, `new:york`.

The above read request is sent to a Cassandra node, which will return an array of columns associated with the column family identified by `new:york`. This array is retrieved from the combination of two sources: `memtable`, and one or more `sstables`. More precisely (Figure 1.9),

- first, `memtable` is scanned;
- second, `sstable` will be searched for row key `new:york` using bloom filter to determine efficiently whether or not `new:york` exists in `sstable`. If `new:york` exists, data to be retrieved will be located by row index.

Cassandra architecture is so designed to optimize read performance for data retrieval.

Figure 1.9
Read on one node



b. In the cluster

Let’s consider a cluster of 8 nodes numbered from 1 to 8 with replicator factor N = 3.

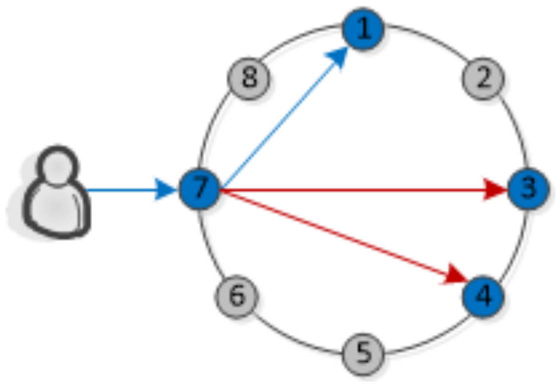
A data read request to retrieve a whole column family whose row key is, for example, `new:york`, is sent by a client to a node in the cluster, say, node 7. N = 3 alive replica nodes (1, 3 and 4) will be sorted by proximity thanks to `snitch`. Node 7 will act as `StorageProxy` and must perform the following tasks (see Figure 1.10):

- sends the read request to retrieve the column family identified by `new:york` to the nearest node, say, node 1;
- sends digest requests to other nodes;

Then, node 1 will return the whole `new:york` column family, i.e., actual data. Node 3 and 4 return a `digest data` which is a hash of the columns, their values, timestamps and other meta-data. Node 7 will compare digest data with actual data. If there is a perfect match, Cassandra will not perform `read repair`. If digest and actual data do not match, Cassandra has to read full column family from digest replicas and make update to out-of-date replicas based on `timestamp`. This mechanism always runs in the background to ensure data consistency in the cluster.

Consistency level sets the number of successful reads before sending query results to the client. The possible values are: ONE, TWO, THREE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL. For instance, if the consistency level is preset to:

Figure 1.10
Read on cluster



- ONE: query results returned from node 1 is sufficient.
- QUORUM: for N = 3, (3 / 2) + 1 = 2 replica nodes will be requested to read actual data. This means that either node 3 or 4 must also read actual data.
- ALL: nodes 1, 3, and 4 will be requested to read actual data. Then node 7 will compare and take the most recent data as query results. If one of three nodes crashes, the read operation fails.

Normally, Cassandra depends on OS to cache `sstable` files. Hence, we can speed up read operations by adding RAM to store information that is in use or used most recently, and enabling the various caches that Cassandra has.

Memory pressure is frequently encountered during read and write operation if Cassandra is not configured appropriately. When such error occurs, a warning message displays with the instructions to solve that incident.

```
WARN 18:58:49,862 Heap is 0.7574948846581447 full.  You may need to
reduce memtable and/or cache sizes. Cassandra will now flush up to the
two largest memtables to free up memory.  Adjust
flush_largest_memtables_at threshold in cassandra.yaml if you don't
want Cassandra to do this automatically
 WARN 18:58:49,862 Flushing CFS(Keyspace='helen', ColumnFamily='ngram')
to relieve memory pressure
```

1.4.6 Delete in Cassandra

Let’s assume that a delete query contains the row key `new:york` of a column family to be deleted from the database.

As `sstable` is immutable, Cassandra cannot simply remove a `new:york` column family from the `sstable`. The following operations will be carried out instead:

- Cassandra updates the column family value, i.e., each value in the array of columns, with a special value called `tombstone`.
- Later, the column family identified by `new:york` will be removed after a `compaction` process which runs at predefined time interval. Two situations may occur during a delete request:

Situation 1: One replica node crashes. Other remaining replica nodes have `tombstone` value in `new:york` column family.

Situation 2: One replica node is down longer than the predefined `compaction` time interval `gc_grace_seconds` (default: 10 days). Data repair must be performed by administrators on every node in the cluster to prevent to `tombstone` data to reappear.

1.5 Solr

As Jonathan Ellis said in his interview about '*Integrating Enterprise Search with Analytics*' on April 16, 2012, Solr is a gold standard for search.

Solr is an open source enterprise search platform based on the powerful Lucene Search Library. Solr's major features include "*power full-text search, hit highlighting, faceted search, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling, and geospatial search*" (Apache Solr).

- Web services: Solr is a Java-based web application, but the users don't need to have any knowledge about Java, and can invoke Lucern Search in any programming language.

- Faceting: A facet represents a specific perspective on content that is typically clearly bounded and mutually exclusive. The combination of all facets and values are often called a faceted taxonomy. Faceted search allows users to customize their own custom navigation by combining various perspectives. Take an example of a tourist who would like to plan his vacation trip in Vietnam: "Destination" facet (values: Sud, Middle, South), "Duration" facet (values: 3, 7, 14 days), "Transportation" facet (values: by air, by car, by train). He could combine values from different facets to drill into search results.

- Easy configuration: Only two XML files, schema.xml and solrconfig.xml, must be edited in order to declare fields to be indexed and their characteristics.

Administration interface: It facilitates the routine tasks such as data loading, index replication, monitoring, logging and cache management

2 Environment Setup and Tests

2.1 Introduction

In this master thesis, a full-text index on Cassandra will be created, then its performance will be compared to the index of Solr search engine. The full-text index on Cassandra will use two indexing techniques, n-gram and reverse.

To this end, an environment must be set up on a computer running Ubuntu, which consists of four tasks:

- Task 1: Check for correct version of Java.
- Task 2: Install Tomcat web server.
- Task 3: Install Solr search engine on Tomcat web server.
- Task 4: Install Cassandra

2.2 Task 1: Check for correct version of Java

Solr is written in Java. Hence, Java Runtime is a prerequisite. For Solr 1.4, we must ensure that the installed Java version be at least 1.6. The following check shows that our Java version 1.7.0 is correct.

```
martin@ubuntu:~$ java -version
java version "1.7.0"
Java(TM) SE Runtime Environment (build 1.7.0-b147)
Java HotSpot(TM) Server VM (build 21.0-b17, mixed mode)
```

2.3 Task 2: Install Tomcat web server

Solr is a component installed on a web server which could be Jetty, Resin or Tomcat. The latter was selected for this research project.

From the terminal under Ubuntu, the command `sudo apt-get install tomcat7` will install Tomcat 7.0. After installation, the following command is issued with administrator privilege to open `tomcat-user.xml` for editing.

```
martin@ubuntu:/etc/tomcat7$ sudo gedit tomcat-users.xml
```

The above XML file must now be edited to enable the manager login as user "`tomcat`" with password "`tomcat`" (insecure). To this end, the appropriate lines must be uncommented by removing `<!--` and `-->` . Of course, usernames and passwords can be added or modified as needed.

```
<!--
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="manager,admin"/>
-->
```

Finally, Tomcat must be restarted for the changes to take effect:

```
martin@ubuntu:/etc/init.d$ sudo ./tomcat7 restart
```

Our Tomcat web server is now ready for use.

Note: The installation process of Tomcat creates a directory `/tomcat7` under `/etc`, and a number of standard subdirectories under `/etc/tomcat7`.

2.4 Task 3: Install Solr search engine on Tomcat web server

During the course of this master project, two versions of Solr were released. There was practically no support for those versions. On the Net, Solr users community do exchange experiences to solve installation and configuration problems. However, most solutions are not applicable to a specific environment. One must proceed and learn by trial and error.

Therefore, Task 3 will describe two completely different installation and configuration procedures based on the author's own experience with the Beta and Official releases.

2.4.1 Install Solr Beta (apache-solr-4.0.0-BETA)

There are two ways to package webapps for deployment in Tomcat, using either the system instance or your own instance. In this project, the system instance was chosen to deploy Solr in order to benefit from webapp autodeployment by Tomcat, i.e., once started, Tomcat will automatically pick up Solr.

Let's assume that two Solr webapps must be created: one for development and another for production, i.e., two instances of Solr side-by-side. Furthermore,

- the Solr installation directory will be `/usr/share/solr`,

- and during the installation process, the deployment descriptor files (`solrdev.xml` and `solrprod.xml`) will be placed in `/etc/tomcat7/Catalina/localhost`.

▪ **Step 1: download Solr**

The latest version of Solr search engine can be downloaded at http://www.apache.org/dyn/closer.cgi/lucene/solr/. Then, the downloaded product is decompressed by executing successively `gunzip` and `tar xvf` into a temporary directory `apache-solr-4`.

After decompression, a WAR file called `apache-solr-4.0.0-BETA.war` must be present in directory `/apache-solr-4/dist`. For ease of use during Solr installation, it is recommended to shorten `apache-solr-4.0.0-BETA.war` to `solr.war`.

- Step 2: create Solr directory trees

In order to create development and production Solr webapps, two directory trees are now created manually under `/etc/tomcat7`:

```
/etc/tomcat7/data/solr

/etc/tomcat7/data/solr/prod
/etc/tomcat7/data/solr/prod/collection1
/etc/tomcat7/data/solr/prod/collection1/data
/etc/tomcat7/data/solr/prod/collection1/conf

/etc/tomcat7/data/solr/dev
/etc/tomcat7/data/solr/dev/collection1
/etc/tomcat7/data/solr/dev/collection1/data
/etc/tomcat7/data/solr/dev/collection1/conf
```

As shown above, within `/data/solr`, two Solr locations with identical directory structure are defined: one for development (`/dev`), another for production (`/prod`). `/collection1` contains two subdirectories `data` and `conf`. While `conf` contains configuration files of a given instance, the `data` subdirectory will be used by Solr to write indexes and logs.

- Step 3: move `solr.war` to a shared location

The `solr.war` file is copied from the temporary directory `/apache-solr-4/dist` into `/usr/share/solr`:

```
martin@ubuntu:~/Downloads/apache-solr-4/dist$ cp solr.war
/usr/share/solr/solr.war
```

`/usr/share/solr` is a shared directory from which `solr.war` file will be invoked by as many instances as needed, e.g., by a development instance and a production instance.

- Step 4: create two descriptor files

The two deployment descriptor files (`solrdev.xml` and `solrprod.xml`) must be created in system-wide instance at location `/etc/tomcat7/Catalina/localhost` as specified at the beginning of section 2.4.1.

First, open the directory `/etc/tomcat7/Catalina/localhost`:

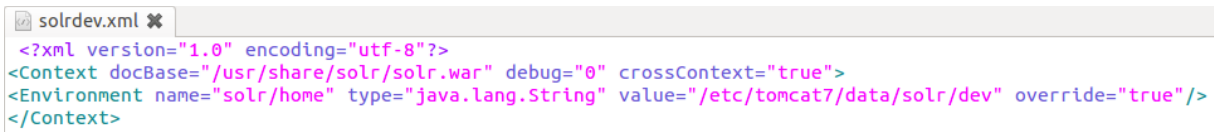
```
martin@ubuntu:~$ cd /etc/tomcat7/Catalina/localhost
```

create solrdev.xml for the development instance

```
martin@ubuntu:/etc/tomcat7/Catalina/localhost$ gksudo gedit solrdev.xml
```

- line 2: solrdev Tomcat Context fragment points docBase to the shared directory (see Step 3) `/usr/share/solr/solr.war`

- line 3: in the environment, the location of `solr/home` is `/etc/tomcat7/data/solr/dev`.



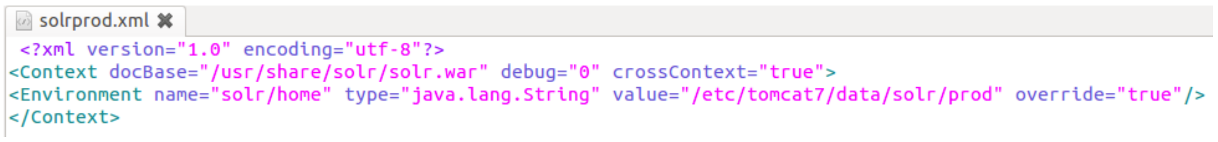
```
<?xml version="1.0" encoding="utf-8"?>
<Context docBase="/usr/share/solr/solr.war" debug="0" crossContext="true">
<Environment name="solr/home" type="java.lang.String" value="/etc/tomcat7/data/solr/dev" override="true"/>
</Context>
```

create `solrprod.xml` for the production instance

```
martin@ubuntu:/etc/tomcat7/Catalina/localhost$ gksudo gedit
solrprod.xml
```

- line 2: solrdev Tomcat Context fragment points docBase to the shared directory (see Step 3) `/usr/share/solr/solr.war`

- line 3: in the environment, the location of `solr/home` is `/etc/tomcat7/data/solr/prod`.



```
<?xml version="1.0" encoding="utf-8"?>
<Context docBase="/usr/share/solr/solr.war" debug="0" crossContext="true">
<Environment name="solr/home" type="java.lang.String" value="/etc/tomcat7/data/solr/prod" override="true"/>
</Context>
```

- Step 5: set write permission on Solr output directories

All results (indexes, logs) will be output by Solr webapp into `/collection1/data` of production and development instances, respectively. Therefore, write permission must be set on `/etc/tomcat7/data/solr/prod/collection1/data` and `/etc/tomcat7/data/solr/dev/collection1/data` by the following commands:

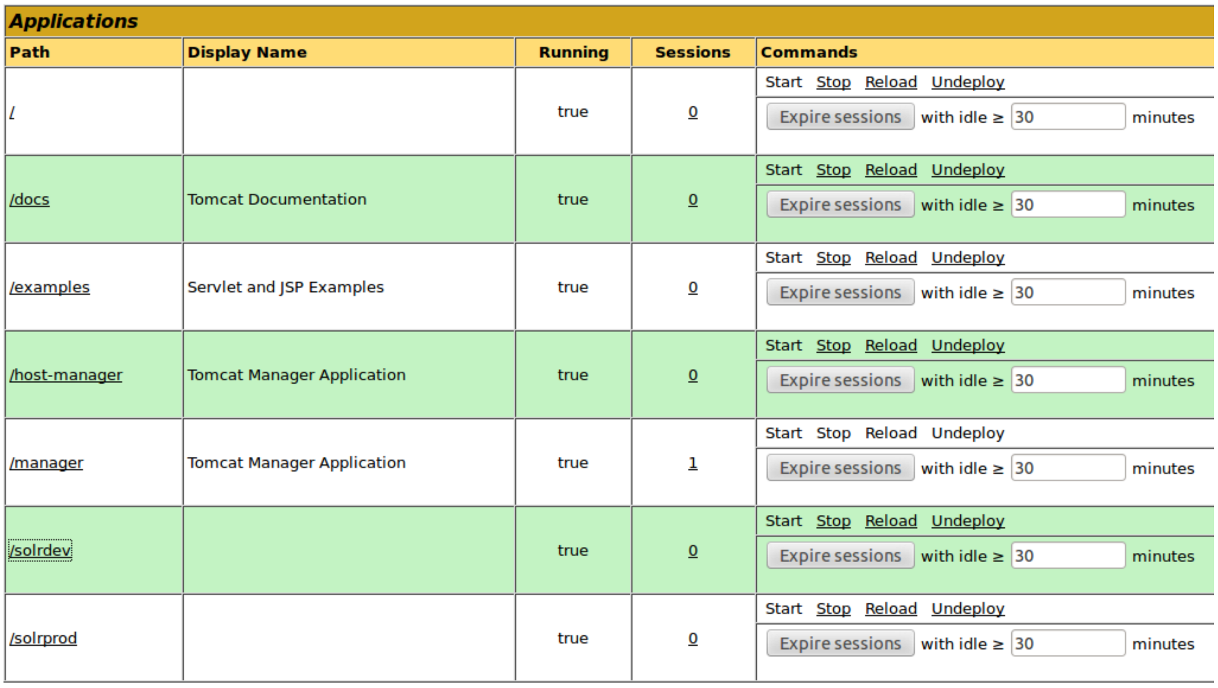
```
martin@ubuntu:~$ sudo chmod 757 -R
/etc/tomcat7/data/solr/prod/collection1/data
martin@ubuntu:~$ sudo chmod 757 -R
/etc/tomcat7/data/solr/dev/collection1/data
```

- Step 6: launch Solr

Restart Tomcat. Solr instances will be picked up automatically by Tomcat as system instance.

If we now access the Tomcat Web Application Manager, we will see two paths, `/solrdev` and `/solrprod`, from which applications may be launched in either development or production instance (Figure 2.11).

Figure 2.11
Tomcat Web Application Manager



Applications				
Path	**Display Name**	**Running**	**Sessions**	**Commands**
/		true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/docs	Tomcat Documentation	true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/examples	Servlet and JSP Examples	true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/host-manager	Tomcat Manager Application	true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/manager	Tomcat Manager Application	true	1	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/solrdev		true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes
/solrprod		true	0	Start Stop Reload Undeploy / Expire sessions with idle ≥ 30 minutes

- Step 7: Configure production and development instances

Of course, the configuration may be performed by the brute force approach through manual editing several required files in XML, TXT, JavaScript, and so on. However, time can be saved by using the sample configuration files bundled with the Solr distribution and located at `/example/solr/conf` in the temporary download directory `/apache-solr-4`.

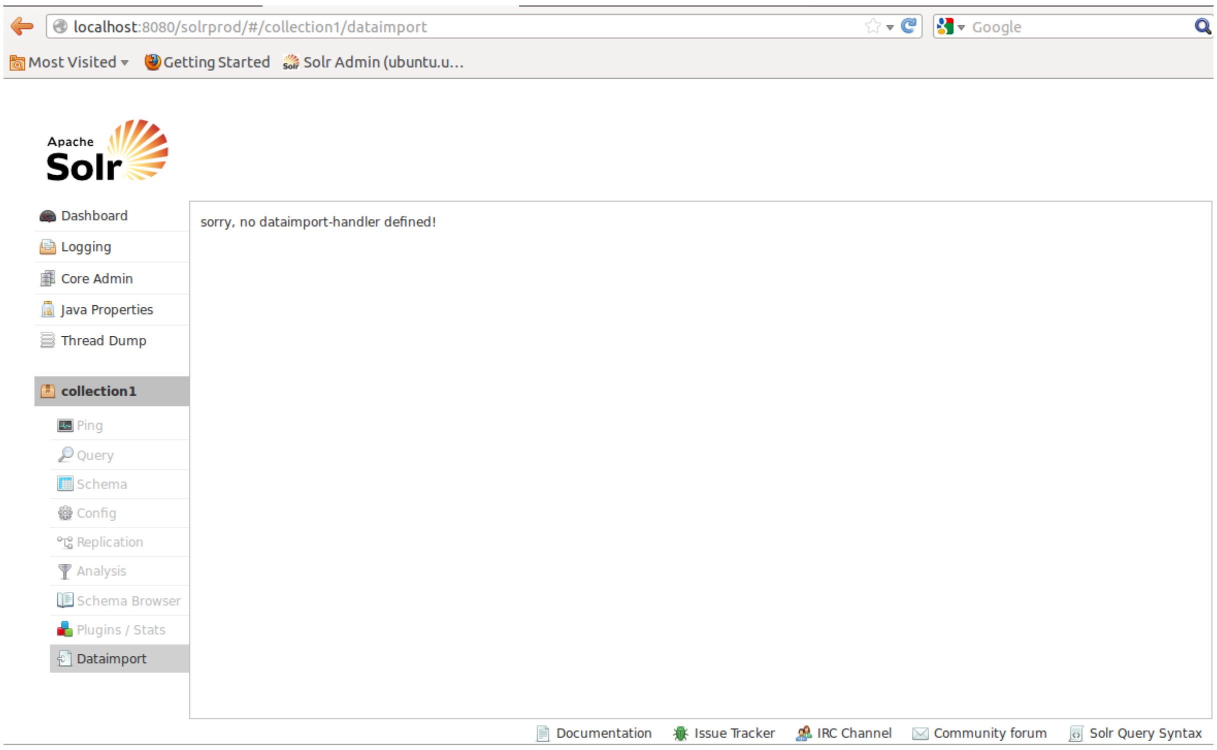
There are two important sample configuration files required to run an instance, `schema.xml` and `solrconfig.xml`.

The following sequences of commands are used to copy the sample configuration files into the destination directories of development and production instances, respectively.

```
martin@ubuntu:~$ cp -R * /etc/tomcat7/data/solr/dev/conf
martin@ubuntu:~$ cp -R * /etc/tomcat7/data/solr/prod/conf
```

Once copied, we now have the basic configuration files which must be then edited to fulfill our needs. After restarting Tomcat with the proper configuration files, Solr home page will be displayed as shown in Figure 2.12.

Figure 2.12
Solr home page



sorry, no dataimport-handler defined!

2.4.2 Install Solr Official (apache-solr-4.2.1)

- Step 1: Download Solr

The latest version of Solr search engine can be downloaded at http://lucene.apache.org/solr/downloads.html. Then, the downloaded product is decompressed by executing successively `gunzip` and `tar xvf` into a temporary directory apache-solr-4.2.1. After decompression, a WAR file called `apache-solr-4.2.1.war` must be present in directory /apache-solr-4.2.1/dist.

- Step 2: Deploy apache-solr-4.2.1.war into Tomcat7 web server

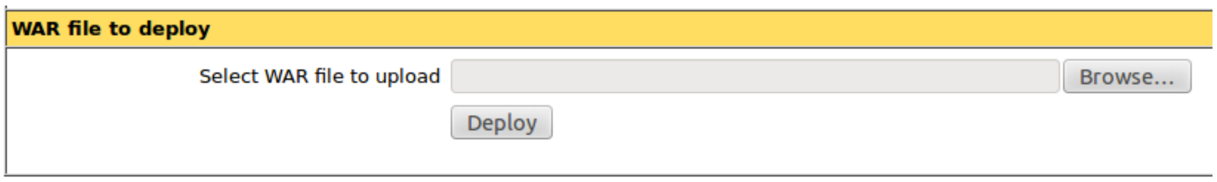
- First, launch the Tomcat Web Application Manager at localhost:8080/manager/html.

- Go to WAR file to deploy under Deploy section (Figure 2.13)

- In the field Select WAR file to upload, click <browse> to navigate to the /apache-solr-4.2.1/dist directory, and choose `apache-solr-4.2.1.war`.

- Click <Deploy>

Figure 2.13
WAR file to deploy



Upon successful deployment, Tomcat manager displays a new line giving the following information under `Applications` section:

- Path: `/apache-solr-4.2.1`
- Display Name:
- Running: `true`
- Sessions: `0`
- Commands: `Start, Stop, Reload, Undeploy`

▪ Step 3: Set up Solr home

- Navigate to the Tomcat directory `$TOMCAT_PATH`, i.e., `/usr/local/apache-tomcat-VERSION`.

- The original file `$TOMCAT_PATH/webapps/apache-solr-4.2.1/WEB-INF/web.xml` contains the following section:

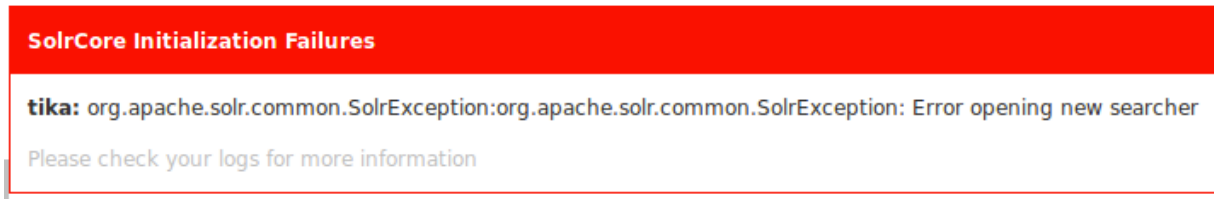
```
<!--
<env-entry>
   <env-entry-name>solr/home</env-entry-name>
   <env-entry-value>/put/your/solr/home/here</env-entry-value>
   <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
-->
```

- Edit this section as follows:
(a) remove the comment lines `<!--` and `-->`
(b) replace `/put/your/solr/home/here` by your Solr home, i.e., for this research: `$SOLR_HOME=/usr/local/apache-solr-4.2.1/example/example-DIH/solr`.

▪ Step 4: Configure Solr instance

Normally, clicking on the path `/apache-solr-4.2.1` under the Applications section of Tomcat Web Application Manager will navigate to Solr home page. However, since configuration is not yet done, this will result in `SolrCore Initialization Failures`,as shown in Figure 2.14.

Figure 2.14
SolrCore Initialization Failures


SolrCore Initialization Failures

tika: org.apache.solr.common.SolrException:org.apache.solr.common.SolrException: Error opening new searcher

Please check your logs for more information

Configuring Solr instance involves editing three files: `solrconfig.xml`, `schema.xml`, and `tika-data-config.xml`.

a. solrconfig.xml

`solrconfig.xml` contains most of configuration parameters for Solr.

```
  <lib dir="../../../../contrib/extraction/lib" />
  <lib dir="../../../../contrib/dataimporthandler/lib/" regex=".*jar$"
/>
  <lib dir="../../../../dist/" regex="apache-solr-dataimporthandler-
.*\.jar" />
```

Edit the original version by replacing all occurrences `../../../../` with absolute path `/usr/local/apache-solr-4.2.1/` which contains subfolders `/contrib` and `/dist`

Let's note that `solrconfig.xml` contains another important section for data import. This section refers to an xml file `tika-data-config.xml` which does not work in our environment. Therefore, it will be created from scratch later in section (c) below.

```
  <requestHandler name="/dataimport"
class="org.apache.solr.handler.dataimport.DataImportHandler">
    <lst name="defaults">
         <str name="config">tika-data-config.xml</str>
    </lst>
  </requestHandler>
```

b. schema.xml

`schema.xml` contains all of the details about which fields documents that will be imported can contain, and how those fields should be dealt with when adding documents to the index, or when querying those fields.

In this research, the xml documents which will be imported have the following simple structure:

```
<FILE>
<DOC>
<DOCNO> … </DOCNO>
<TEXT> … <TEXT>
</DOC>
</FILE>
```

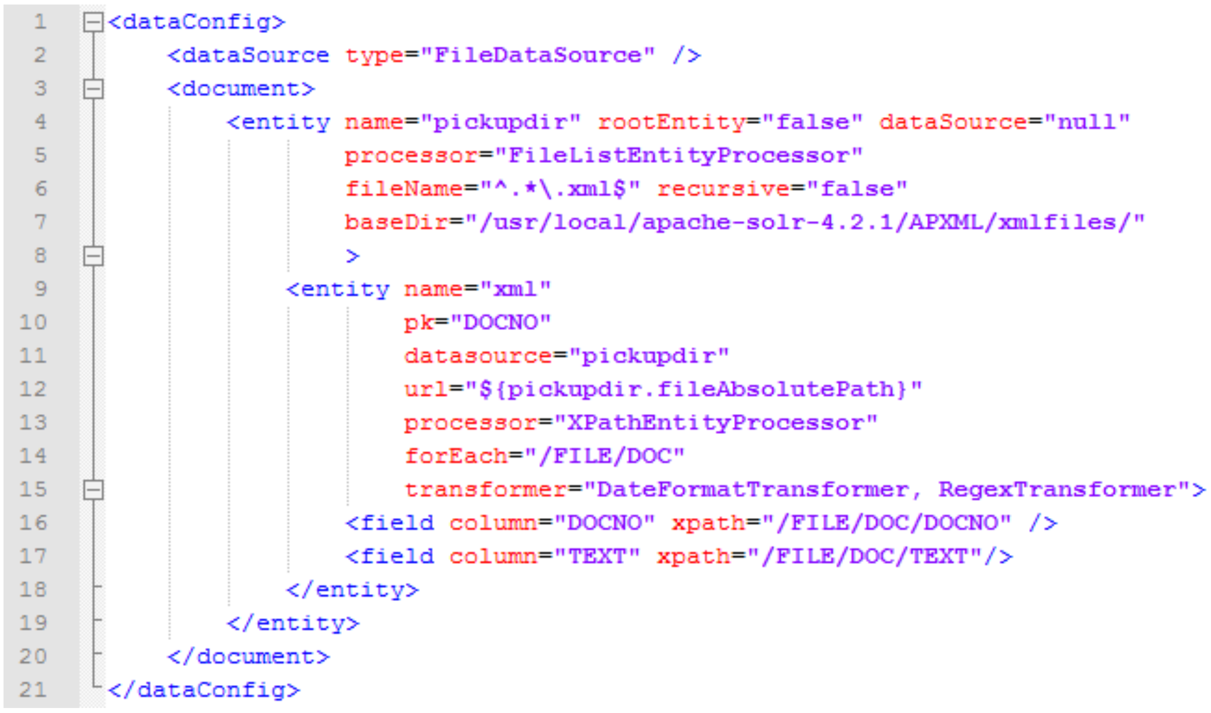
Therefore, `schema.xml` must be edited to include `DOCNO`, `TEXT` as fields and `TEXT` as default search field.

```
<fields>
   <field name="DOCNO" type="string" indexed="true" stored="true"/>
   <field name="TEXT" type="text" indexed="true" stored="true" />
</fields>
 <!-- field for the QueryParser to use when an explicit fieldname is
absent -->
 <defaultSearchField>TEXT</defaultSearchField>
```

c. tika-data-config.xml

The following listing shows the contents of `tika-data-config.xml` created from scratch, since the original version only works for specific purposes different from this research.

Figure 2.15
Tika-data-config.xml



```
<dataConfig>
    <dataSource type="FileDataSource" />
    <document>
        <entity name="pickupdir" rootEntity="false" dataSource="null"
                processor="FileListEntityProcessor"
                fileName="^.*\.xml$" recursive="false"
                baseDir="/usr/local/apache-solr-4.2.1/APXML/xmlfiles/"
                >
            <entity name="xml"
                    pk="DOCNO"
                    datasource="pickupdir"
                    url="${pickupdir.fileAbsolutePath}"
                    processor="XPathEntityProcessor"
                    forEach="/FILE/DOC"
                    transformer="DateFormatTransformer, RegexTransformer">
                <field column="DOCNO" xpath="/FILE/DOC/DOCNO" />
                <field column="TEXT" xpath="/FILE/DOC/TEXT"/>
            </entity>
        </entity>
    </document>
</dataConfig>
```

Figure 2.15 shows Tika-data-config.xml file that indices Solr server connect to datasource as explained below:

- line 2: **FileDataSource**: use to read from local files. The file is read with the default platform encoding. It can be overridden by specifying the encoding in solrconfig.xml.

- line 5: **FileListEntityProcessor**: An EntityProcessor instance which can stream file names found in a given base directory matching patterns and returning rows containing file information.

It supports querying a give base directory by matching:

- regular expressions to file names
- excluding certain files based on regular expression
- last modification date (newer or older than a given date or time)
- size (bigger or smaller than size given in bytes)
- recursively iterating through sub-directories

Its output can be used along with FileDataSource to read from files in file systems.

- line 6: File type is XML file.

- line 7: The XML input files are located at `/usr/local/apache-solr-4.2.1/APXML/xmlfiles/`

- line 10: The primary key for the entity is DOCNO. It is optional and only needed when using delta-imports.

- line 13: **XPathEntityProcessor**: uses a streaming xpath parser to extract values out of XML documents. It is typically used in conjunction with FileDataSource.

- line 14: `/FILE/DOC` demarcates a record.

- line 16 and line 17: two elements will be read.

2.5 Task 4: Install Apache Cassandra database management system

2.5.1 Install apache-cassandra-1.0.12

This version is released on 4th October 2012.

Step 1: Perform pre-installation operations

It is necessary to upgrade Ubuntu system software in order to avoid some conflicts caused by Cassandra new installation. The following commands are issued sequentially to upgrade the software.

```
martin@ubuntu:~$ sudo apt-get update
martin@ubuntu:~$ sudo apt-get upgrade
```

Then, Ubuntu must be told where to look for the Cassandra installation package, since the package is not in the standard Ubuntu repositories. To this end,

- open `sources.list` which provides information about the location of packages for Ubuntu

```
martin@ubuntu:~$ gksudo gedit /etc/apt/sources.list
```

- append the following two lines to `sources.list`.

```
deb http://www.apache.org/dist/cassandra/debian 10x main
deb-src http://www.apache.org/dist/cassandra/debian 10x main
```

The `10x` argument tells APT (**A**dvanced **P**ackaging **T**ool) to install the 1.0.x branch of Cassandra. The latest Cassandra release is 1.1.6. To install this version, the `.../debian 10x main` must be change to `.../debian 11x main`.

- save `sources.list`

- run `sudo apt-get update` to update the system software. An error will absolutely occur with the following message giving an important information: the GPG Public Key value.

```
W: GPG error: http://www.apache.org unstable Release: The following
signatures couldn't be verified because the public key is not available:
NO_PUBKEY F758CE318D77295D
```

- issue the following three commands to add the repository's GPG Public Key

```
martin@ubuntu:~$ gpg --keyserver wwwkeys.eu.pgp.net --recv-keys
F758CE318D77295D
martin@ubuntu:~$ sudo apt-key add ~/.gnupg/pubring.gpg
martin@ubuntu:~$ sudo apt-get update
```

Step 2: Install Cassandra

```
martin@ubuntu:~$ sudo apt-get install cassandra
```

Step 3: Configure Cassandra (single node)

In order to avoid conflicts with other programs, the port number and host of Cassandra could be changed by editing the file `cassandra.yaml`.

```
martin@ubuntu:/etc/cassandra$ gksudo gedit cassandra.yaml
```

Change `listen_address`, `rpc_address` and `seeds` to 134.21.245.168 which must be the same as IP Address of the computer where Cassandra is installed. Also, `rpc_port` could be changed if needed.

```
listen_address: 134.21.245.168
rpc_address: 134.21.245.168
- seeds: 134.21.245.168
rpc_port: 9160
```

Step 4: Start Cassandra

Cassandra should be restarted for the changes to take effect:

(a) start Cassandra as a Service

```
martin@ubuntu:/etc/init.d$ sudo ./cassandra start
```

(b) start Cassandra as a Stand-Alone process

```
martin@ubuntu:$ cd $CASSANDRA_HOME
martin@ubuntu:$ sh bin/cassandra -f
```

Cassandra version can be checked by the command `/bin/cassandra-cli`. In the following, the actual Cassandra version is 1.0.12 corresponding to the argument `10x` that was set before Cassandra installation (see Step 1).

```
martin@ubuntu:/bin$ cassandra-cli
Welcome to Cassandra CLI version 1.0.12

Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown]
```

When connecting to Cassandra, there are two frequent problems that cause the connection to fail (`connection refused` error):

- the previous Cassandra process has not been killed
- the `cassandra.yaml` file has not been correctly edited (see Step 3)

The following command could be used to verify whether Cassandra is listening on the right address.

```
martin@ubuntu:/etc/init.d$ netstat -ant | grep 9160
tcp        0      0 134.21.245.168:9160     0.0.0.0:*
LISTEN
```

The above response shows that Cassandra is listening on the address `134.21.245.168`, port number `9160` that was set in Step 3.

Step 5: Stop Cassandra

To stop the Cassandra process, find the Cassandra Java process ID (PID), and then kill -9 that process using its PID. There are several ways to find a PID, e.g., via JConsole. For example, the following command kills the process with PID 1539.

```
martin@ubuntu:$ kill -9 1539
```

2.5.2 Configure Cassandra multi-node cluster

a. Terminology: multi-node cluster and related concepts

The planning, configuration and deployment of a multi-node cluster require a thorough understanding of important related concepts which will be briefly discussed in this section.

A *multi-node cluster* is a group of one or more connected *nodes*.

In Cassandra, each *data center* has a name and can span one or more nodes in the cluster. A data center is also a *replication group*.

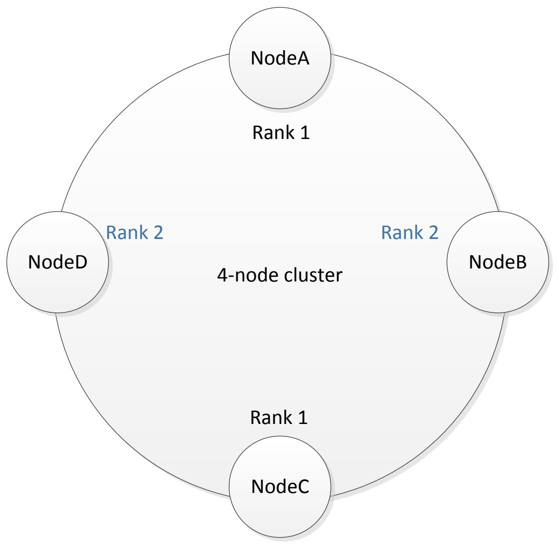
A number of related stable nodes will be chosen as *seeds*. Each data center should have more than one seed node. At least one seed from the seed node list must be contacted by each newly created node to be able to find each other and to allow information exchange between nodes using Gossip mechanism.

The *rack* configuration allows Cassandra to optimize replica placement for better fault tolerance. For example, the rack configuration can tell Cassandra to keep replicas on different racks, so that, if one rack is down, data will not be lost.

A *snitch* defines how nodes are grouped into data centers and racks for write and read operations and informs Cassandra about the network topology for efficient routing of requests. Furthermore, a *snitch* allows Cassandra to distribute replicas among racks. There are three available snitches: SimpleSnitch, RackInferringSnitch and PropertyFileSnitch.

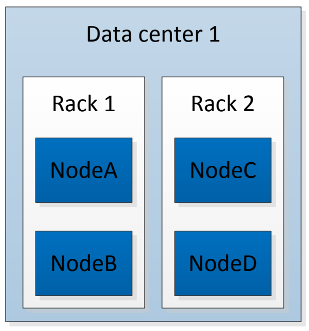
Each node in the cluster owns a part of *token* range from 0 to 2^127-1. If the Nth node in the cluster has token value T(N), the node owns range from T(N-1)+1 to T(N). Cassandra decide nodes where a data should be stored based on the consistent mapping of the row key and token range. (ref: http://wiki.apache.org/cassandra/GettingStarted)

Figure 2.16
Token



The *ring* is a range mapping formed by the server token values which are stored and wrap around.

Figure 2.17
Nodes, Data centers and Racks



b. Configure a 4-node cluster with a single data center

- Step 1: Collect information on the cluster and network topology

First, we will configure a multi-node cluster, then data centers will be defined, giving a multi-data center cluster. To this end, the following cluster characteristics must be defined and collected:

- Cluster name: ?
- How many nodes the cluster will have?
- Node names?
- How many nodes per data center (or replication group)?
- Seed nodes?
- The IP address of each node?
- The token for each node?
- The chosen snitch?

For example, a 4-node cluster must be configured spanning 2 racks in a single data center.

- The cluster name is `4-node Cluster`
- The cluster has 4 nodes: `nodeA, nodeB, nodeC, nodeD`
- The single data center has 4 nodes.
- There are two seed nodes: `nodeA` and `nodeC`
- The IP addresses of nodes as the following:

```
nodeA: 192.168.0.20 (seed1)
nodeB: 192.168.0.12
nodeC: 192.168.0.22 (seed2)
nodeD: 192.168.0.10
```

- The token for each node will be calculated later in Step 2
- The `SimpleSnitch` is chosen for our single data center deployment

Step 2: Calculate 4 tokens for the new nodes in the cluster

Each Cassandra node is assigned a token at startup. A token can be set either by assigning explicitly a value to the `initial_token` property in the `cassandra.yaml` configuration file, or automatically via the bootstrap process.

The explicit value of `initial_token` is calculated by carrying out the following four operations:

(a) Create a blank file called `tokengentool` which stands for "**token gen**erator **tool**"

```
vi tokengentool
```

(b) Insert the following Python statements into `tokengentool`. This piece of program requires the user to enter the number of nodes in the cluster. The program then generates the values of `initial_token` from 0 to 2^127-1 which will be displayed on screen.

```
#! /usr/bin/python
import sys
if (len(sys.argv) > 1):
    num=int(sys.argv[1])
else:
    num=int(raw_input("How many nodes are in your cluster? "))
for i in range(0, num):
    print 'token %d: %d' % (i, (i*(2**127)/num))
```

(c) Save and close the file, then make it executable by the following `chmod` command:

```
martin@ubuntu:$ chmod +x tokengentool
```

(d) Run tokengentool:

```
./tokengentool
```

When the question "*How many nodes are in your cluster?*" displays, enter the total number of nodes in the cluster, e.g., **4** in our example. The computed values of tokens are listed on screen as below:

```
token 0: 0
token 1: 42535295865117307932921825928971026432
token 2: 85070591730234615865843651857942052864
token 3: 127605887595351923798765477786913079296
```

Note that the value of the first token is always *zero*.

Step 2: Configure each node in the 4-node cluster

The configuration parameters are defined in `cassandra.yaml`. This is an important configuration file which must be present for each node at location `$CASSANDRA_HOME/conf/`. In this step, `cassandra.yaml` is edited to include cluster configuration data collected in the previous steps. Any change to this file requires the node to be restarted. The following four examples give some main configuration parameters of NodeA, NodeB, NodeC and NodeD excerpted from their respective `cassandra.yaml`.

- The seed provider maintains a comma-delimited list of hosts (contact points) that Cassandra nodes use to find each other and learn the topology of the ring.

- The listen address indicates IP address or associated hostname that other Cassandra nodes use to connect to this node.

- The rpc address is for remote procedure calls from client connections.

NodeA:

```
cluster_name: ‘4-node Cluster’
initial_token: 0
seed_provider:
        - seeds: "192.168.0.20,192.168.0.22"
listen_address: 192.168.0.20
rpc_address: 192.168.0.20
```

```
endpoint_snitch: SimpleSnitch
```

NodeB:

```
cluster_name: '4-node Cluster'
initial_token: 127605887595351923798765477786913079296
seed_provider:
       - seeds: "192.168.0.20,192.168.0.22"
listen_address: 192.168.0.12
rpc_address: 192.168.0.12
endpoint_snitch: SimpleSnitch
```

NodeC:

```
cluster_name: '4-node Cluster'
initial_token: 85070591730234615865843651857942052864
seed_provider:
       - seeds: "192.168.0.20,192.168.0.22"
listen_address: 192.168.0.22
rpc_address: 192.168.0.22
endpoint_snitch: SimpleSnitch
```

NodeD:

```
cluster_name: '4-node Cluster'
initial_token: 42535295865117307932921825928971026432
seed_provider:
       - seeds: "192.168.0.20,192.168.0.22"
listen_address: 192.168.0.10
rpc_address: 192.168.0.10
endpoint_snitch: SimpleSnitch
```

Step 3: Start Cassandra 4-node cluster

There are two ways to start/restart the cluster: as a Service or as a Stand-Alone process.

(a) issue the following command on each node to start Cassandra as a Service

```
martin@ubuntu:/etc/init.d$ sudo ./cassandra start
```

(b) issue the following command on each node to start Cassandra as a Stand-Alone process

```
martin@ubuntu:$ cd $CASSANDRA_HOME
martin@ubuntu:$ sh bin/cassandra -f
```

Step 4: Check node health

At any time, the health of each cluster node could be checked either via `JConsole` which has a user-friendly GUI, or via the `nodetool` utility which is a command line interface. Since `JConsole` consumes a significant amount of system resources, `nodetool` utility is preferred and explained in detail below. The readers who are interested in `JConsole` could visit http://www.datastax.com/docs/1.0/operations/monitoring.

Let's recall that `$CASSANDRA_HOME` is set to `/usr/local/apache-cassandra-1.1.6` in our configuration. In `$CASSANDRA_HOME/bin`, there are many utilities for cluster management, SSTable monitoring, node checking, etc.

The `nodetool` utility provides commands for viewing table metrics, server metrics, and compaction statistics. The utility also includes decommissioning a node, repair ring node, and moving partitioning tokens.

A given node is considered healthy if the command `nodetool info` executes successfully by displaying all parameter values of that node as shown below, otherwise the command fails with an error message `Connection refused`.

```
root@ubuntu:/usr/local/apache-cassandra-1.1.6/bin# nodetool info
Token            : 82077673235279609097885452317579849321
Gossip active    : true
Thrift active    : true
Load             : 2.43 GB
Generation No    : 1368717669
Uptime (seconds) : 18943
Heap Memory (MB) : 399.05 / 2008.00
Data Center      : datacenter1
Rack             : rack1
Exceptions       : 0
Key Cache        : size 672 (bytes), capacity 104857584 (bytes), 17 hits,
19 requests, 0.846 recent hit rate, 14400 save period in seconds
Row Cache        : size 0 (bytes), capacity 0 (bytes), 0 hits, 0
requests, NaN recent hit rate, 0 save period in seconds
```

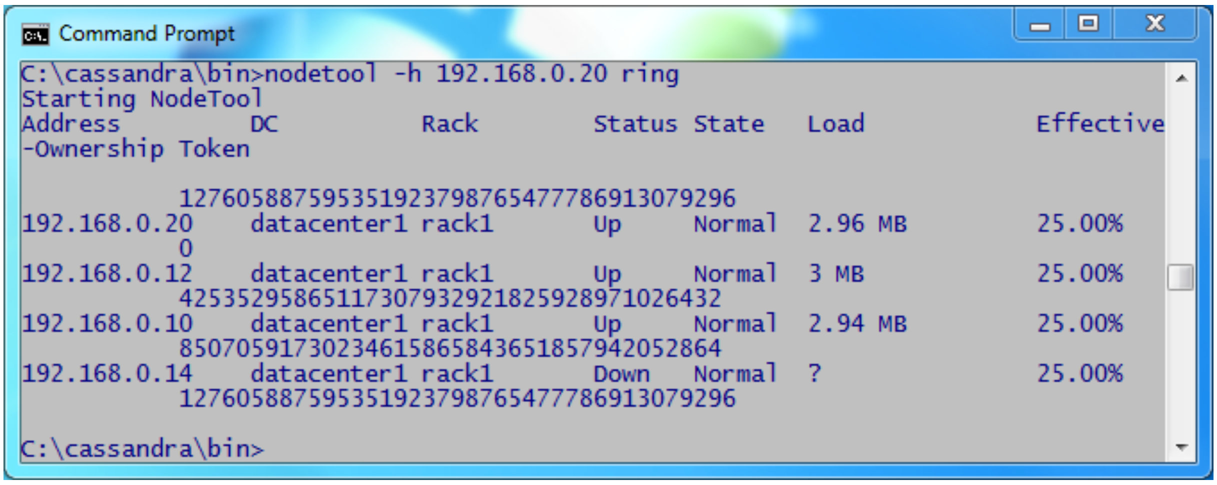
The `nodetool` utility can also be used to check the health of the ring as shown in Figure 2.18.

- `Status` column shows whether or not that node is available.

- The `Effective-Ownership` column indicates the percentage of the ring (`keyspace`) handled by that node.

- As a ring, the first and the last token are the same.

Figure 2.18
Parameters of a ring when a node dies



```
C:\cassandra\bin>nodetool -h 192.168.0.20 ring
Starting NodeTool
Address         DC          Rack        Status State   Load            Effective
-Ownership Token

             127605887595351923798765477786913079296
192.168.0.20    datacenter1 rack1       Up     Normal  2.96 MB         25.00%
             0
192.168.0.12    datacenter1 rack1       Up     Normal  3 MB            25.00%
             42535295865117307932921825928971026432
192.168.0.10    datacenter1 rack1       Up     Normal  2.94 MB         25.00%
             85070591730234615865843651857942052864
192.168.0.14    datacenter1 rack1       Down   Normal  ?               25.00%
             127605887595351923798765477786913079296

C:\cassandra\bin>
```

2.6 Task 5: Put the system to work

2.6.1 Query with Solr

The test will be carried out in three steps:

Step 1: import a large volume of data about 10’000 scientific papers: author, paper’s content, year of publication, etc.

Step 2: make queries using search criteria based on keywords or any character strings.

Step 3: record search performance data into an Excel file for later comparison with Cassandra performance.

Step 1: import data

- Raw data about 10'000 scientific papers are downloaded from a server of the Department of Informatics (`diuflx09.unifr.ch`), then converted into XML files of 1'000 records each. A record contains data pertaining to one scientific paper.

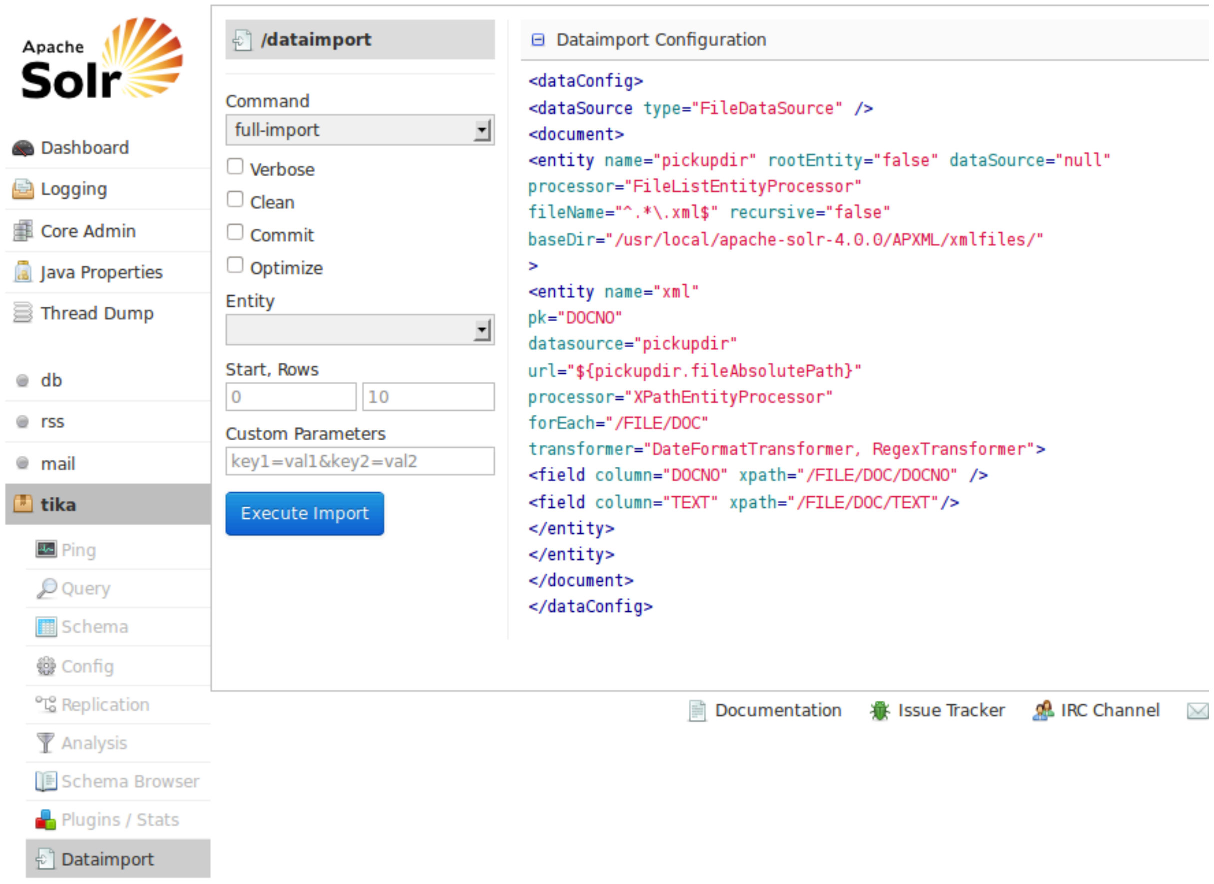
- Move those XML files into `/usr/local/apache-solr-4.2.1/APXML/xmlfiles/`. This directory must be the same as defined by the value of `baseDir` in `tika-data-config.xml` (see Section 2.4.2.).

- In the Apache Solr navigator (Figure 2.19), scroll to `tika` core, then click on `Dataimport` to display the Dataimport interface.

- Choose `full-import` command from the popup menu which offers two alternatives (`full-import`, `delta-import`).

- Finally, click on `<Execute Import>` blue button, and wait until import be completed.

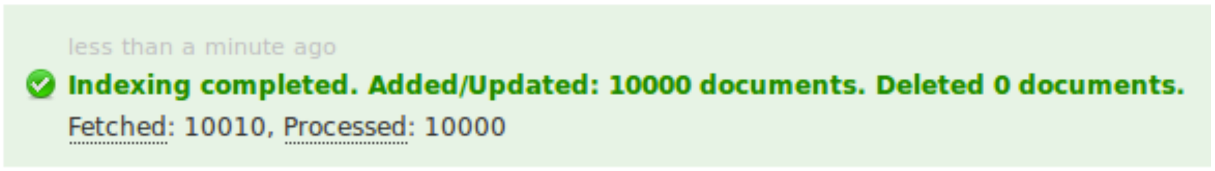
Figure 2.19
Dataimport interface



```
<dataConfig>
<dataSource type="FileDataSource" />
<document>
<entity name="pickupdir" rootEntity="false" dataSource="null"
processor="FileListEntityProcessor"
fileName="^.*\.xml$" recursive="false"
baseDir="/usr/local/apache-solr-4.0.0/APXML/xmlfiles/"
>
<entity name="xml"
pk="DOCNO"
datasource="pickupdir"
url="${pickupdir.fileAbsolutePath}"
processor="XPathEntityProcessor"
forEach="/FILE/DOC"
transformer="DateFormatTransformer, RegexTransformer">
<field column="DOCNO" xpath="/FILE/DOC/DOCNO" />
<field column="TEXT" xpath="/FILE/DOC/TEXT"/>
</entity>
</entity>
</document>
</dataConfig>
```

- During data import, an index is created from the 10'000 records. Figure 2.20 shows that 10'000 records (documents) from 10 files (Fetched 10'010 - 10'000 = 10) were processed.

Figure 2.20
Indexing results



```
less than a minute ago
Indexing completed. Added/Updated: 10000 documents. Deleted 0 documents.
Fetched: 10010, Processed: 10000
```

Step 2: make query

As shown in Figure 2.21:

- Enter the search string “`per capita alcohol consumption`” into the field labeled 'q'. Note that by default, the line `<solrQueryParser defaultOperator="OR"/>` in `schema.xml` tells Solr that it must search "`per OR capita OR alcohol OR consumption`". Of course, the `defaultOperator` can be changed to "`AND`" manually.

- Select `xml` as output format in the field labeled '`wt`'. Other proposed formats are `json`, `python`, `ruby`, `php`, `csv`.

- Click <`Execute Query`> blue button.

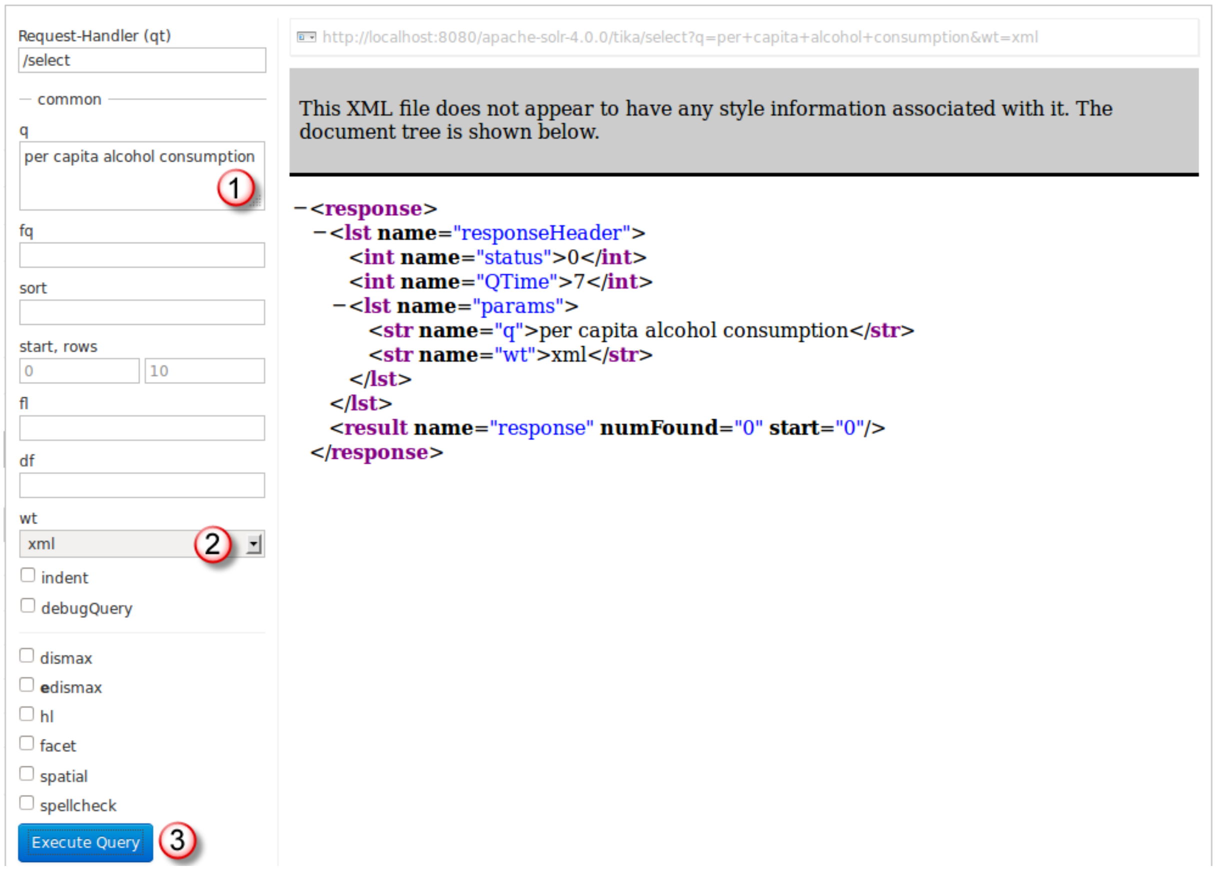
Solr displays the document tree of an XML file resulting from the search, and containing two interesting information:

- The line `<int name="QTime">7</int>` tells us that the query execution takes 7 milliseconds.

- The line `<result name="response" numFound="0" start="0"/>` tells us that Solr did not find any occurrence of "`per capita alcohol consumption`".

For the author who knows that there is at least one occurrence of that search string, the above query result is evidently incorrect. By experience with Solr, the author also observes that this error systematically occurs at every first query.

Figure 2.21
Make query: no occurrence of search string found



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
−<response>
  −<lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">7</int>
    −<lst name="params">
        <str name="q">per capita alcohol consumption</str>
        <str name="wt">xml</str>
      </lst>
    </lst>
    <result name="response" numFound="0" start="0"/>
  </response>
```

To get the correct result, the following work-around must be carried out, then the same query must be repeated.

Work-around step 1: remove the files in directory /data/index

```
root@ubuntu:/usr/local/apache-solr-4.2.1/example/example-
DIH/solr/tika/data/index# rm *
```

Work-around step 2: restart Tomcat

```
root@ubuntu:/usr/local/apache-tomcat-7.0.27/bin# sh shutdown.sh
root@ubuntu:/usr/local/apache-tomcat-7.0.27/bin# sh startup.sh
```

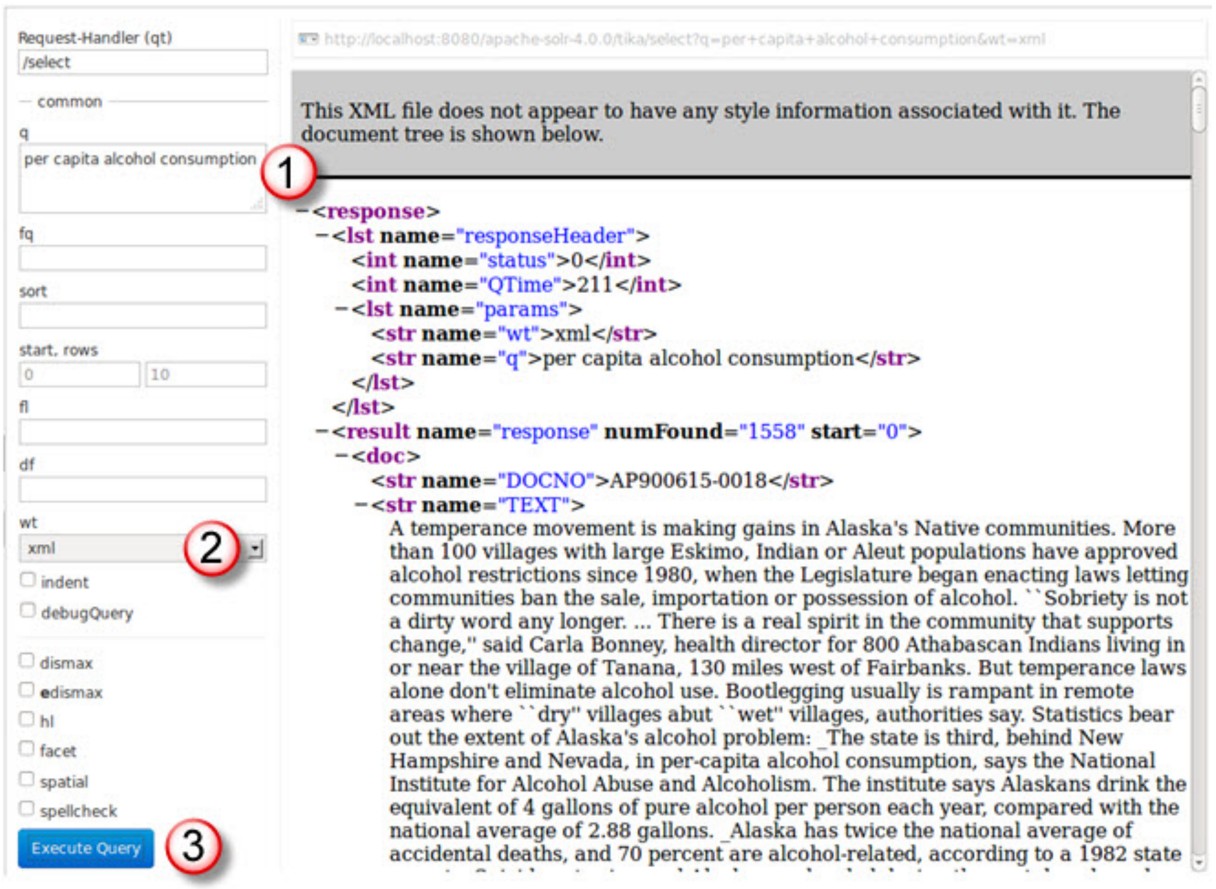
Work-around step 3: check the presence of required files after restarting Tomcat

These files define together the index used by Solr to execute queries.

```
root@ubuntu:/usr/local/apache-solr-4.2.1/example/example-
DIH/solr/tika/data/index# ls
_0.fdt  _0_Lucene40_0.frq  _0_Lucene40_0.tip  _0.si
_0.fdx  _0_Lucene40_0.prx  _0_nrm.cfe         segments_2
_0.fnm  _0_Lucene40_0.tim  _0_nrm.cfs         segments.gen
```

Now, repeat the query with search string "`per capita alcohol consumption`". This time, Solr returns the correct XML document tree (Figure 2.22) that contains 1558 scientific papers out of 10'000. The query execution takes 211 milliseconds to complete.

Figure 2.22
Make query: 1558 occurrences found



http://localhost:8080/apache-solr-4.0.0/tika/select?q=per+capita+alcohol+consumption&wt=xml

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<response>
  -<lst name="responseHeader">
     <int name="status">0</int>
     <int name="QTime">211</int>
   -<lst name="params">
      <str name="wt">xml</str>
      <str name="q">per capita alcohol consumption</str>
     </lst>
   </lst>
  -<result name="response" numFound="1558" start="0">
   -<doc>
      <str name="DOCNO">AP900615-0018</str>
    -<str name="TEXT">
        A temperance movement is making gains in Alaska's Native communities. More
        than 100 villages with large Eskimo, Indian or Aleut populations have approved
        alcohol restrictions since 1980, when the Legislature began enacting laws letting
        communities ban the sale, importation or possession of alcohol. ``Sobriety is not
        a dirty word any longer. ... There is a real spirit in the community that supports
        change,'' said Carla Bonney, health director for 800 Athabascan Indians living in
        or near the village of Tanana, 130 miles west of Fairbanks. But temperance laws
        alone don't eliminate alcohol use. Bootlegging usually is rampant in remote
        areas where ``dry'' villages abut ``wet'' villages, authorities say. Statistics bear
        out the extent of Alaska's alcohol problem: _The state is third, behind New
        Hampshire and Nevada, in per-capita alcohol consumption, says the National
        Institute for Alcohol Abuse and Alcoholism. The institute says Alaskans drink the
        equivalent of 4 gallons of pure alcohol per person each year, compared with the
        national average of 2.88 gallons. _Alaska has twice the national average of
        accidental deaths, and 70 percent are alcohol-related, according to a 1982 state
```

Step 3: record search performance

Table 2.1 shows the performance of query execution. The first line records the performance of searching 1'000 scientific papers. The same query was repeated 15 times, giving 15 observed execution times from which the average in milliseconds (`AVG`) and the standard deviation (`STDEVP`) were computed.

The same experiment was carried out with increasing number of scientific papers: 2'000, 3'000, …, 10'000.

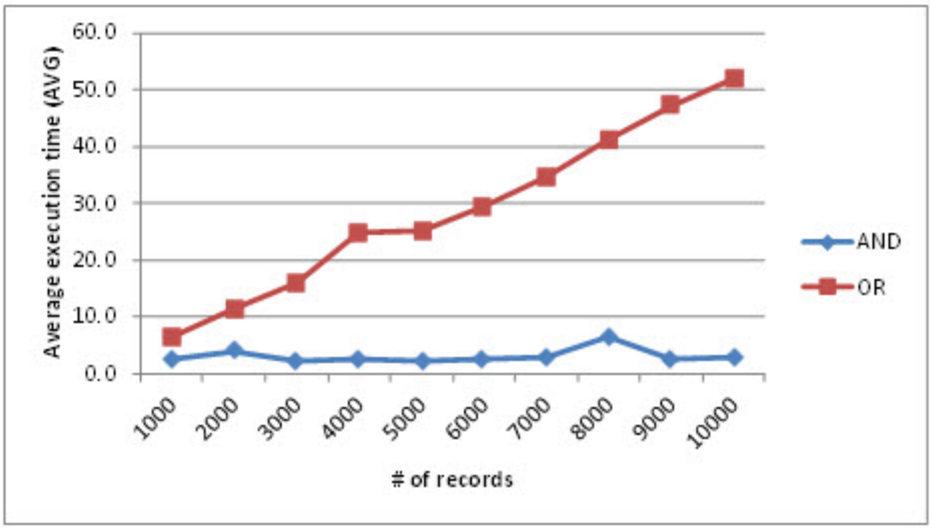
With the "`AND`" operator, the average execution time is significantly lower than that with the "`OR`" operator. With both operators, the standard deviation indicates wide dispersion about the average, due to caching mechanism. For example, for the 15 experiments with 1'000 records, the cache is initially empty before the first experiment; from the second experiment onwards, read access to database is no longer necessary, thanks to the cache contents which can be efficiently reused.

Cache management is critical to query performance and needs time to gain experience in customizing parameters. This aspect is outside the scope of this research project.

Table 2.1
Performance of query execution

# of records	AND		OR	
	AVG	STDEVP	AVG	STDEVP
1000	2.6	5.7	6.6	5.7
2000	4.1	10.2	11.4	7.2
3000	2.4	3.9	16.1	5.5
4000	2.4	4.9	24.8	22.4
5000	2.3	5.0	25.2	7.1
6000	2.6	4.9	29.4	5.4
7000	3.0	6.1	34.7	8.0
8000	6.6	21.7	41.2	6.6
9000	2.5	5.2	47.3	10.9
10000	2.9	5.9	52.1	8.4

Figure 2.23
Average execution time



2.6.2 Query with Cassandra

a. Create keyspace, column family

Cassandra data modelling is based on the concepts of Column, Super Column, Row, Column Family, Keyspace:

"The keyspace is the top level unit of information in Cassandra. Column families are subordinate to exactly one keyspace. While variations exist, all queries for information in Cassandra take the general form get(keyspace, column family, row key)." (Featherston, 2010; Sadalage, 2013).

This section presents the three steps to create a keyspace and column families of our data model which will be translated into a Cassandra database:

- Step 1: connect to Apache Cassandra
- Step 2: create keyspace
- Step 3: create column families
- Step 4: insert data
- Step 5: make query

▪ **Step 1: connect to Apache Cassandra**

- Start Apache Cassandra (see Step 4 in Section 2.5.1).

- Launch the command line interface.

`bin/cassandra-cli` is an interactive command line interface for Cassandra. Run the following commands to launch it:

```
martin@ubuntu:$ cd $CASSANDRA_HOME/bin
martin@ubuntu:$ cassandra-cli
```

A connection with Apache Cassandra is then etablished and the interface is ready to accept the commands. As shown below, the connection with Cassandra on "`Test Cluster`" is successful at IP address `127.0.0.1` on port `9160`.

```
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 1.0.12

Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown]
```

▪ **Step 2: create a keyspace**

The `create keyspace` command will be used to declare a new keyspace called `helen`. The keyspace name `helen` was chosen as one of the characters in the Trojan War of Greek mythology (Cassandra, Hector, Menelaus, Helen, etc.). `helen` will store data about scientifc papers grouped into column families. The following properties define Cassandra Replication (Feipeng, 2012).

- `NetworkTopologyStrategy` is used since our application is deployed to multiple (2) racks.

- The `replication_factor` determines the total number of replicas. Here, the value 2 means data are committed to 2 separate nodes, and also that each node now acts as a replica for 2 separate token ranges.

```
create keyspace helen
    with placement_strategy = 'org.apache.cassandra.locator.
NetworkTopologyStrategy'
    and strategy_options = {replication_factor:2};
```

The `use` command means that subsequent commands will apply to the keyspace `helen`.

```
use helen;
```

Note: A semicolon (';') is required to end each command.

- **Step 3: create column families**

Within the `helen` keyspace, 10 column families will be created using the `create column family` command. Each column family consists of a given number of scientific papers for the purpose of querying Cassandra and measuring its performance as a function of column family size.

The first column family named `1000records` stores 1'000 scientific papers, the second named `2000records` stores 2'000 scientific papers:

```
[default@helen] create column family 1000records and comparator =
'AsciiType';
[default@helen] create column family 2000records and comparator =
'AsciiType';
```

Similar commands must be entered to create the remaining column families. The last one is:

```
[default@helen] create column family 10000records and comparator =
'AsciiType';
```

- **Step 4: insert data**

There are at least two ways to insert data into Cassandra:

- Using `cassandra-cli`: for each scientific paper, one must insert not only information about the paper (author, tittle, contents, date of publication, and so on), but also, for each n-gram string in the paper's contents, its number of occurrences and the associated document ID. For example, the following command insert the unigram string `capita` into column family `1000records`, which occurs 20 times in the scientific paper whose ID is `SP12398`.

```
set 1000records['capita']['SP12398'] = 20;
```

There will be as many such insertions as n-gram strings (unigram, bigram and trigram) in the paper. This method is very time consuming because of the large number of n-gram strings to be indexed for a given document, hence although technically possible, it is practically unfeasible.

- Using a Java program specifically coded for data insertion. This program aims at two purposes:

    - read data about scientific papers from XML input files
    - count the number of occurrences of each n-gram string in the paper's contents associated with a document ID.

Each XML input file contains a chunk of 1'000 scientific papers. For each experiment, the exact number of XML files to be read by the Java program is determined by the desired size of the experiment's dataset (1'000, 2'000, …, 10'000 records).

This method is clearly much more efficient than the previous one thanks to the automated data input and counting of occurrences of n-gram string.

The design of this Java program will be presented later in part b of Section 2.6.2.

- **Step 5: make query**

To compare the performance between Solr and Cassandra, a query using the same search string "`per capita alcohol consumption`" will be tested. For better efficiency, we will develop a special piece of Java program to make queries against Cassandra database instead of using `cassandra-cli` or `cassandra-cql`.

This piece of program is included in the query module of the Java program presented in part b of Section 2.6.2.

b. Develop a Java program for index creation and queries

This program consists of two modules: the index creation module and the query module.

- **Functionalities of the index creation module**

The data insertion module performs the following operations in Cassandra:

- read data about scientific papers from an XML input file;
- store the data according to the document structure defined by the XML tags (`DOCNO`, `FILEID`, …, `TEXT`);
- split the contents of `TEXT` into n-gram strings (unigram, bigram and trigram), e.g., `per`, `capita`, `alcohol`, `consumption`, `per capita`, `capita alcohol`, `alcohol consumption`, `per capita alcohol`, `capita alcohol consumption`.
- remove stopwords and perform stemming;
- create an inverted index to store the n-gram strings (excluding stopwords) and associated document IDs;
- load the inverted index into Cassandra.

- **Functionalities of the query module**

The query module performs the following operations:

- split the search string "`per capita alcohol consumption`" into n-gram substrings;
- remove stopwords and perform stemming;
- search Cassandra database for scientific papers using n-gram substrings as keywords;
- write search results into output file.

The query module can include several queries with different search strings to be executed in a single run.

3 Experimentation and Comparison of Search Query Performance

3.1 Objectives

According to more or less recent statistics (e.g., Putnam C., 2010; McGee M. 2012), Facebook's users create 3.2 billion 'LIKE' and comments every day, upload 2.5 billion photos each month, along with other data such as video, blog notes, maps, etc.. Once logged in, a Facebook member can search, for example, his friend by last name among billions of people. When a member uploads a picture of his last summer holidays on his wall, Facebook searches that member's friends in order to inform them about his upload. These two examples show that searching is an important operation in social networks such as Facebook. Of course, searching is a basic operation of search engines, too.

The search operation must deal with big volume of data efficiently. Therefore, the performance of indexing techniques plays a critical role in the efforts to reduce processing time.

In this master thesis, a full-text index in Apache Cassandra was designed using n-gram model and implemented. Its performance was then compared with another powerful search engine, Apache Solr.

The remaining of this chapter is structured as follows. Section 3.2 explains the index creation module and the search query module coded in Java under Cassandra. Next, the experimentation plan is presented. Then, the performance results will be analyzed based on the search experiments carried out on a Cassandra multi-node cluster. Section 3.3 explains the search query module coded in Java under Solr, and the experimentation plan which is carried out in this environment. Section 3.4 is devoted to the performance comparison between Cassandra and Solr in terms of query execution time. In Section 3.5, the findings of this research project will be summarized.

3.2 Experimentation on Cassandra multi-node cluster

The sample data are records of 28'000 scientific papers. Each record contains one paper's attributes: document number, file id, headline, author, author's company, content of paper, etc.

Prior to experimentation, the sample data are converted into 28 XML documents. Each document groups 1'000 records.

The first experiment will be carried out with one XML document (1'000 records). An n-gram based index is created for this first experiment as described in section 3.6.2 of Chapter 3.

The second experiment will be carried out with two XML documents (2'000 records). An n-gram based index is created for this second experiment.

The third one with three XML documents (3'000 records) and its own n-gram based index will be carried out, and so on.

The twenty-eighth experiment will include 28 XML documents (28'000 records) with its own n-gram based index.

Within each experiment:

- The n-gram based index creation time is collected.

- A search query is then executed 15 times and the execution times measured, from which the average execution time and the standard deviation are computed.

3.2.1 Understanding the index creation module in Java

The index creation module, coded in Java, consists of the steps described in section 2.6.2, Chapter 2. In this section, the Java source code of this module will be explained.

Figure 3.24 is an excerpt from the index creation module in the 10th experiment.

- line 1: The XML input files are located at `/home/thuhang/NetBeansProjects/ngram2cassandra/xmlfiles`

- line 5: The statement `noFiles = 10` declares that there are 10 XML input files for this experiment.

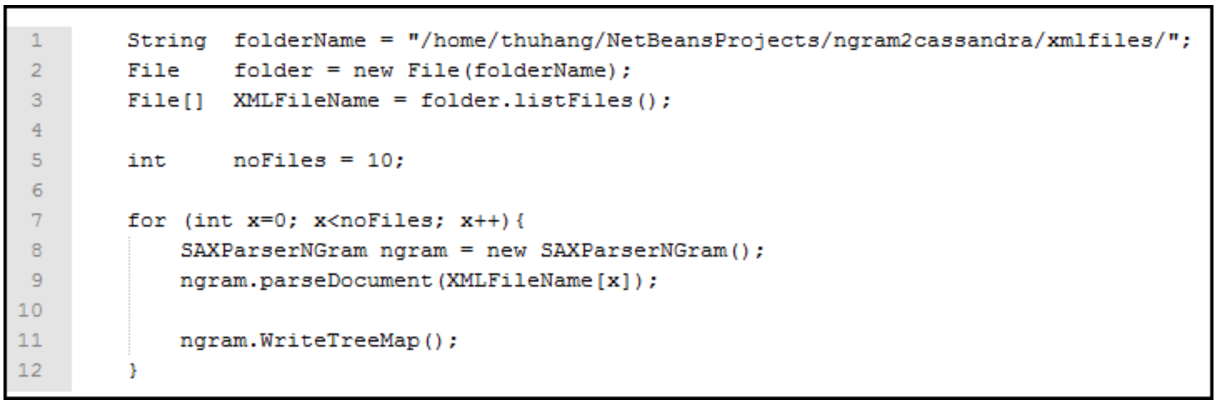
- line 7: The `for` statement specifies that the loop will be executed 10 times, one for each input file.

- line 8: n-gram object is created from `SAXParserNGram` class.

- line 9: The method `parseDocument` of n-gram object has one argument, `XMLFileName[x]`.This method parses the (x+1)th input file and indexes it according to n-gram model with `n = 1, 2, 3`.

- line 11: The resulting index is written into Cassandra database.

Figure 3.24
Index creation module

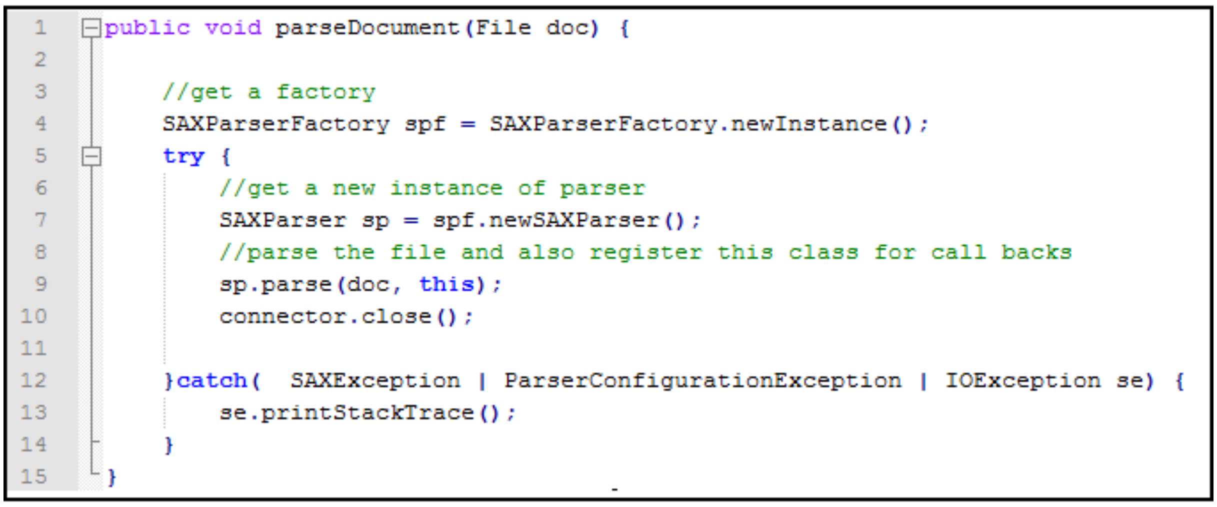


```
String  folderName = "/home/thuhang/NetBeansProjects/ngram2cassandra/xmlfiles/";
File    folder = new File(folderName);
File[]  XMLFileName = folder.listFiles();

int     noFiles = 10;

for (int x=0; x<noFiles; x++){
    SAXParserNGram ngram = new SAXParserNGram();
    ngram.parseDocument(XMLFileName[x]);

    ngram.WriteTreeMap();
}
```

In order to understand the `parseDocument` method of `SAXParserNGram` class, let's start from the SAX's[1] `DefaultHandler` class. This class has methods to receive notifications during an XML file parsing (e.g., end of the document, end of an element, character data inside an element, parser warning, and so on). For our experiments, an XML reader class, `SAXParserNGram`, is developed, which inherits from `DefaultHandler` class. This inheritance allows `SAXParserNGram` to call the `parse` method of `SAXParser`.

Line 9 shows that the `parseDocument` method invokes the `parse` method of `SAXParser`. The argument `this` means `DefaultHandler`. Hence, `SAXParser` will parse the content of an input XML file, `doc`, using `DefaultHandler`.

Figure 3.25
parseDocument method



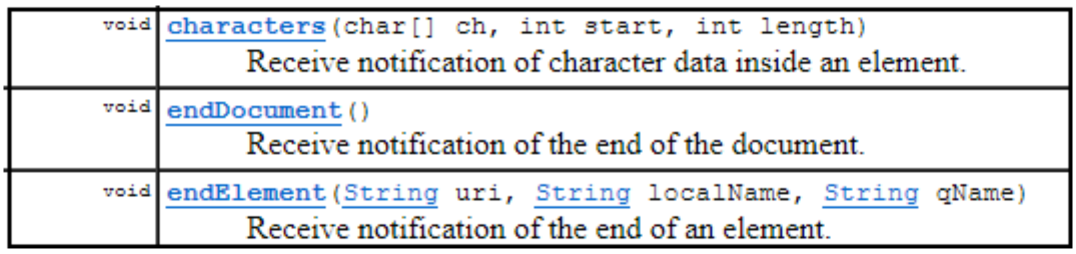
```
1  public void parseDocument(File doc) {
2
3      //get a factory
4      SAXParserFactory spf = SAXParserFactory.newInstance();
5      try {
6          //get a new instance of parser
7          SAXParser sp = spf.newSAXParser();
8          //parse the file and also register this class for call backs
9          sp.parse(doc, this);
10         connector.close();
11
12     }catch(  SAXException | ParserConfigurationException | IOException se) {
13         se.printStackTrace();
14     }
15 }
```

Three methods have been extended in `SAXParserNGram` to allow the retrieval of element contents, attributes, and text contents during parsing: `startElement`, `characters`, and `endElement` (Figure 3.26).

[1] SAX stands for Simple API for XML.

Once retrieved, the element contents, attributes, and text contents will be used to remove stopwords, and build the n-gram based index. The full Java source code is available in the Appendix A.

Figure 3.26
Extended methods



void	characters(char[] ch, int start, int length) Receive notification of character data inside an element.
void	endDocument() Receive notification of the end of the document.
void	endElement(String uri, String localName, String qName) Receive notification of the end of an element.

3.2.2 The performance of index creation

The performance analysis of index creation consists to observe the execution time of the `for` loop described in section 2.1 (Figure 3.24). The observed execution times of ten experiments, for single node cluster, and 28 experiments, for 4-node cluster are listed in Table 3.2 and plotted in Figure 3.27.

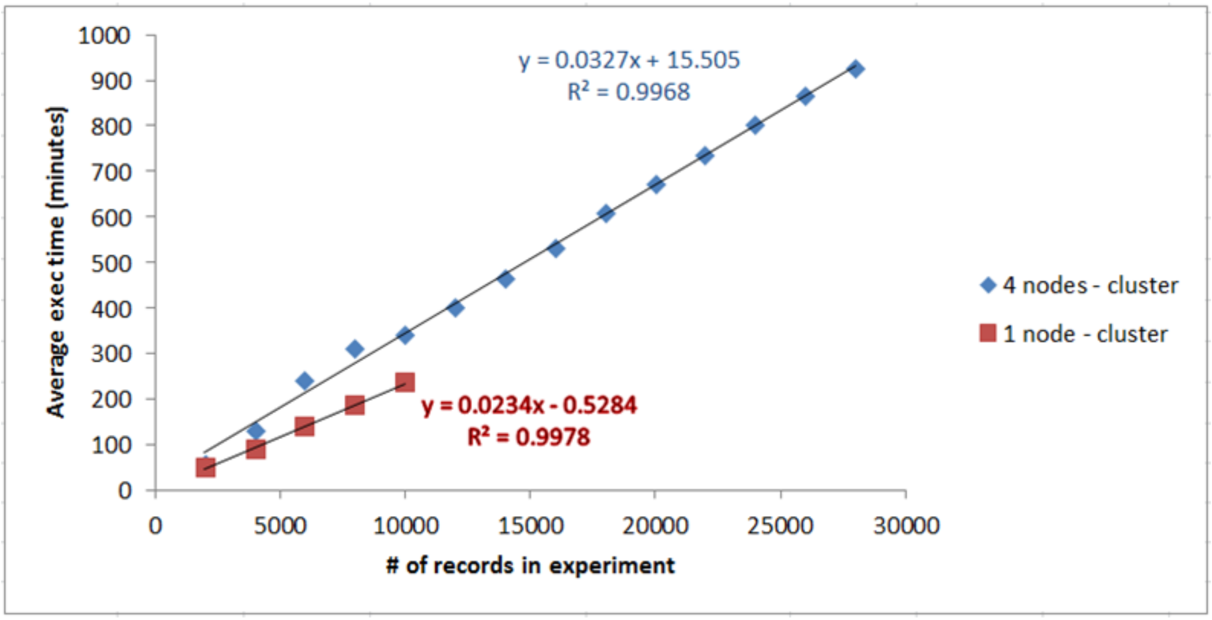
Table 3.2
Average execution time of index creation module

# of records	4 nodes - cluster (minutes)	1 node - cluster (minutes)
2000	55.89	50.58
4000	131.30	88.71
6000	241.23	138.60
8000	310.77	185.46
10000	339.83	236.30
12000	399.40	
14000	463.74	
16000	531.60	
18000	607.52	
20000	670.24	
22000	736.25	
24000	802.69	
26000	864.13	
28000	925.26	

In Table 3.2, as the number of records to be indexed grows, we observe that the average execution time increases in both cluster configurations.

Furthermore, Figure 3.27 shows graphically the average execution times of index creation on 4-node cluster (blue diamond markers) and on single node cluster (red square markers). We observe that both curves follow the same upward trend. In other words, Figure 3.27 suggests that, in both cluster configurations, there is a linear relationship between the number of records and the average execution time. The slope of single node cluster is smaller than the 4-node cluster's one, i.e., for a given number of records in experiment, the index creation on single node cluster takes less time than on the 4-node cluster. This fact can be explained by the mechanism of writing on multi-node cluster. When a write request is sent in one node in the cluster, depending on the predetermined Consistency level, that node has to wait for a number of success write from replicas nodes before sending the success or failure confirmation to the client.

Figure 3.27
Performance of index creation



For the single node cluster, the computed coefficient of determination $R^2 = 0.9978$ shows a strong linear relationship between the number of records and the average execution time. Using the classical linear regression model y = ax + b, we obtain the following estimates:

```
a = 0.0234, b = -0.5284
average execution time = 0.0234 * number of records - 0.5284
```

For the 4-node cluster, the computed coefficient of determination $R^2 = 0.9968$ also shows a strong linear relationship between those two variables. Using the classical linear regression model y = ax + b, we obtain the following estimates:

```
a = 0.0327, b = 15.505
average execution time = 0.0327 * number of records + 15.505
```

The estimated regression coefficient, 0.0327 (minutes), is interpreted as an increase of the average execution time resulting from a unit increase of the number of scientific papers in experiment.

3.2.3 Understanding the search query module in Java

Each experiment consists of processing a sequence of 250 search queries by a Java program. The search strings required by queries are stored in an XML file which consists of 250 XML `top` elements. Each `top` element includes `num`, `title`, `desc`, and `narr` elements. The `title` element is of particular interest to us since it contains the search string for a given query.

Figure 3.28 shows an excerpt of the Java program that forms the queries based on the `title` element as explained below:

- line 10: The 250 queries are stored in `/home/thuhang/NetBeansProjects/ngram2cassandra/queries.xml`.

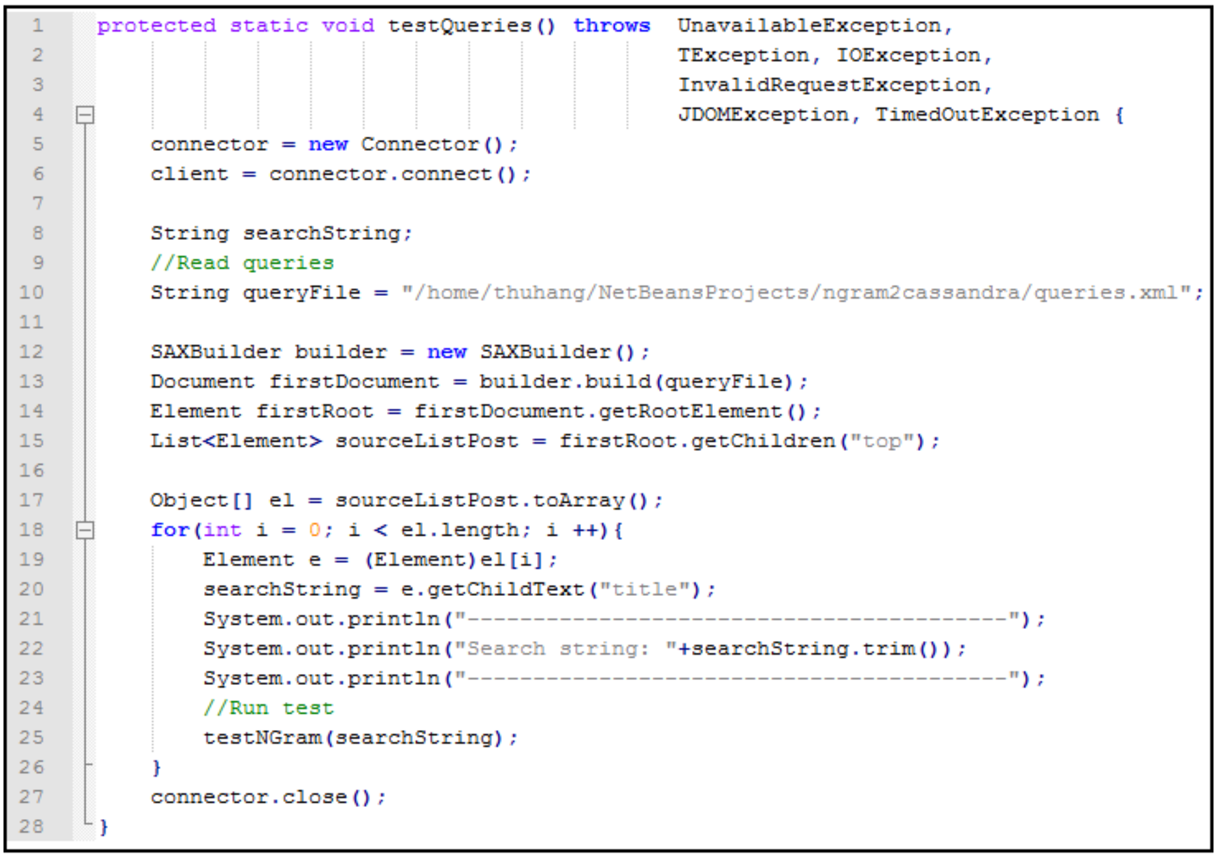
- line 18: The `for` statement specifies that the loop will be executed 250 times, one for each query, i.e., an XML `top` element.

- line 20: The method `getChildText` extracts the search string, i.e., an XML `title` element.

- line 25: The method `testNGram` removes stopwords from the search string, performs stemming, creates unigram, bigram, trigram tokens. Then, `testNGram` builds a query with the OR operator connecting the tokens and another query with the AND operator. Finally, `testNGram` sends the two queries to Cassandra. The full Java source code of `testNGram` is available in the Appendix A.

Cassandra executes the query using its inverted n-gram based index and returns the results to the Java program for display.

Figure 3.28
Java code to form the queries



```java
protected static void testQueries() throws  UnavailableException,
                                            TException, IOException,
                                            InvalidRequestException,
                                            JDOMException, TimedOutException {
    connector = new Connector();
    client = connector.connect();

    String searchString;
    //Read queries
    String queryFile = "/home/thuhang/NetBeansProjects/ngram2cassandra/queries.xml";

    SAXBuilder builder = new SAXBuilder();
    Document firstDocument = builder.build(queryFile);
    Element firstRoot = firstDocument.getRootElement();
    List<Element> sourceListPost = firstRoot.getChildren("top");

    Object[] el = sourceListPost.toArray();
    for(int i = 0; i < el.length; i ++){
        Element e = (Element)el[i];
        searchString = e.getChildText("title");
        System.out.println("-----------------------------------------");
        System.out.println("Search string: "+searchString.trim());
        System.out.println("-----------------------------------------");
        //Run test
        testNGram(searchString);
    }
    connector.close();
}
```

For illustration purposes, Figure 3.29 lists two queries out of 250. In the first query, the search string is `Oscar winner selection`, delimited by `<title>` and `</title>`. In the second query, the search string is `women clergy`.

Figure 3.29
Listing of two queries, delimited by <top> and </top>

```
<query>
    <top>
        <num> Number: 685 </num>
        <title> Oscar winner selection </title>
        <desc> How are Oscar winners selected? </desc>
        <narr>
        Relevant documents describe the process by which Oscar
        winners are determined, including the methods by which
        potential awardees are selected and nominated, the qualifications
        needed to become an eligible voter, and the number of
        eligible voters.
        </narr>
    </top>
    ...
    <top>
        <num> Number: 445 </num>
        <title> women clergy </title>
        <desc> Description:
        What other countries besides the United States are considering
        or have approved women as clergy persons?
        </desc>
        <narr> Narrative:
        To be relevant, a document must indicate either a country
        where a woman has been installed as clergy or a country
        that is considering such an installation.  The clergy position
        must be as church pastor rather than some other church capacity
        (e.g., nun or choir member).
        </narr>
    </top>
    <top>
        ...
    </top>
</query>
```

3.2.4 Experimentation plan

28 experiments will be carried out. The experimentation plan can be seen through the worksheets in Table 3.3,Table 3.4, and Table 3.5.

Each experiment is assigned a predefined number of scientific papers. For example, Experiment no.1 has 1'000 scientific papers, Experiment no.2 has 2'000 scientific papers, Experiment no.10 has 10'000 scientific papers.

Each experiment will process 250 predefined search queries. Each query runs:

- 25 times with Unigram & AND operator,
- 25 times with Bigram & AND operator,
- 25 times with Trigram & AND operator,
- 25 times with Unigram & OR operator,

- 25 times with Bigram & OR operator, and
- 25 times with Trigram & OR operator.

Observed execution times are reported into the shaded areas of the following tables.

Table 3.3
Plan of Experiment no.1

Experiment no.	# of scientific papers	Query no.	Run no.	Unigram AND	Run no.	Bigram AND	Run no.	Trigram AND	Run no.	Unigram OR	Run no.	Bigram OR	Run no.	Trigram OR
1	1000 ... 1000	1	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	1000 ... 1000	2	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	...	...	...		...		...		...		...		...	
	1000 ... 1000	250	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	

Table 3.4
Plan of Experiment no.2

Experiment no.	# of scientific papers	Query no.	Run no.	Unigram AND	Run no.	Bigram AND	Run no.	Trigram AND	Run no.	Unigram OR	Run no.	Bigram OR	Run no.	Trigram OR
2	2000 ... 2000	1	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	2000 ... 2000	2	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	...	...	...		...		...		...		...		...	
	2000 ... 2000	250	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	

Table 3.5
Plan of Experiment no.10

Experiment no.	# of scientific papers	Query no.	Run no.	Unigram AND	Run no.	Bigram AND	Run no.	Trigram AND	Run no.	Unigram OR	Run no.	Bigram OR	Run no.	Trigram OR
10	10000 ... 10000	1	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	10000 ... 10000	2	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	...	...	...		...		...		...		...		...	
	10000 ... 10000	250	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	

Since the index contains unigram, bigram and trigram entries, each query can return several query results. With 10'000 scientific papers and 250 queries, the query results set is quite huge and could be a representative sample for performance study.

Our study consists to compare the performance of Cassandra between unigram-based search, bigram-based search and trigram-based search. To this end, after executing 250 queries according to the experiment plan, those queries are divided into three groups based on the obtained results set, as explained below and represented by Table 3.6:

- **Query group 1**: This group includes those queries which return the DOCNOs of scientific papers containing found unigram tokens only. There are 112 such queries. As shown in row 1 of Table 3.6, 48.21% out of 10'000 scientific papers are found to contain unigram tokens only using search with AND operator, and 97.32% with OR operator.

- **Query group 2**: This group includes those queries which return the DOCNOs of scientific papers containing found unigram and bigram tokens only. There are 115 such queries. As shown in row 2 of Table 3.6, 66.09% out of 10'000 scientific papers are found to contain unigram tokens using search with AND operation, and 100% with OR operator. For bigram tokens found, the percentages are 18.26% and 100%, respectively.

- **Query group 3**: This group includes those queries which return the DOCNOs of scientific papers containing found unigram, bigram and trigram tokens. There are 23 such queries. As shown in row 3 of Table 3.6, 86.96% out of 10'000 scientific papers are found to contain unigram tokens using search with AND operation, and 100% with OR operator. For bigram tokens found, the percentages are 69.57% and 100%, respectively. For trigram tokens found, the percentages are 69.57% and 100%, respectively.

We notice two particularities from Table 3.6, which might be useful for defining search engine's strategy:

- First, the search with OR operator gives higher percentage of hits than with AND operator. This could explain why Google search strategy uses OR operator.

- Second, as shown in rows 2 and 3, the search based on unigram tokens give higher percentage of hits than that based on bigram and trigram tokens. Nevertheless, query results are evidently more relevant with search based on bigram and trigram.

Table 3.6
Percentage of hits by query groups

	# of queries	Unigram		Bigram		Trigram	
		AND	OR	AND	OR	AND	OR
Query group 1	112	48.21%	97.32%	n/a	n/a	n/a	n/a
Query group 2	115	66.09%	100%	18.26%	100%	n/a	n/a
Query group 3	23	86.96%	100%	69.57%	100%	69.57%	100%

During the experiments, execution times of search queries are collected. Since within an experiment, each query execution is repeated 25 times for a given operator, the average execution time, and the standard deviation will be computed based on 25 observations.

In our study of queries' performance, we consider three selected queries, one from each group, and analyze their corresponding execution times.

- Query group 1: The 249th query is selected, which has "`term limits`" as search string.

- Query group 2: The 145th query is selected, which has "`women clergy`" as search string.

- Query group 3: The 247th query is selected, which has "`air traffic controller`" as search string.

3.2.5 The performance of search query

a. Query group 1: 249th query with search string "term limits"

Before examining the observed performance of this query's execution, let's recall how the average execution time and the standard deviation are obtained from the experiments.

Cassandra has two caches: the key cache, and the row cache. In order to observe the effect of caching on query execution time, both caches were emptied before the first run only. Let's consider, for example, the execution of the 249th query in Experiment no.10. Table 3.7 shows the execution times of unigram-based search with AND (column 3) and OR operator (column 4).

Table 3.7
Experiment no.10: 249th query

# of scientific papers	run no.	Exec time in millisecs with AND operator	Exec time in millisecs with OR operator
10000	1	25.842	15.537
10000	2	16.198	17.112
10000	3	16.175	15.652
10000	4	15.716	15.963
...	...	...	...
10000	24	15.989	16.317
10000	25	21.998	21.850
Average		17.611	16.747
Standard deviation		3.159	1.836

For all 28 experiments of the 249th query, Table 3.8 presents the computed average execution times (columns AVG) and standard deviations (columns STD) of unigram-based search with AND and OR operators.

Let's examine the two AVG columns. In both cases, the AVG value increases as the number of records in the experiment augments. The STD columns will be discussed later in connection with query groups 2 and 3 in the next sub-section e.

Table 3.8
All experiments for the 249th query

# of records in experiment	Unigram AND operator		Unigram OR operator	
	AVG	STD	AVG	STD
2000	15.557	3.971	15.063	3.489
4000	44.438	9.664	43.332	7.754
6000	42.645	3.388	45.707	5.254
8000	61.100	4.605	67.730	12.537
10000	70.984	10.062	76.514	8.003
12000	70.753	6.008	74.964	5.877
14000	67.816	4.700	68.180	12.372
18000	96.447	8.421	104.005	24.753
20000	97.504	15.349	111.516	30.159
22000	153.654	102.401	97.276	22.569
24000	81.522	12.498	89.961	13.966
28000	104.403	15.811	114.374	45.066

Figure 3.30
All experiments for the 249th query: Average execution times

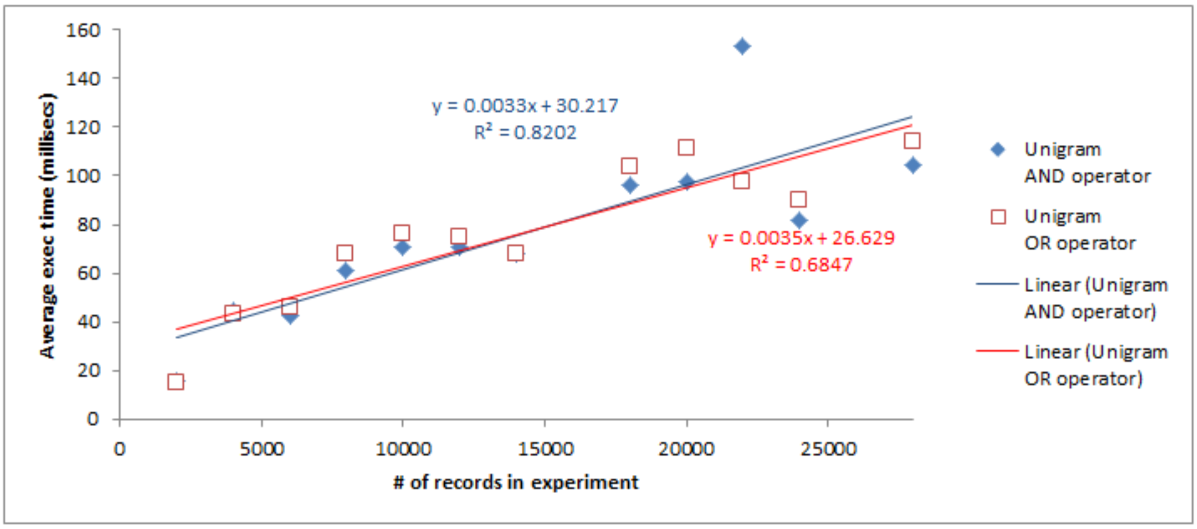


Figure 3.30 shows graphically the average execution times of unigram-based search with operators AND (blue diamond markers) and OR (red square markers). We observe that both curves follow the same behavior. More precisely, Figure 3.30 suggests that there might be a linear relationship between the number of records and the average execution time.

For the case of Unigram with AND operator, the computed coefficient of determination $R^2 = 0.8202$ shows that there is a strong linear relationship between those two variables. Using the classical linear regression model y = ax + b, we obtain the following results:

```
a = 0.0033, b = 30.217
average execution time = 0.0033 * number of records + 30.217
```

For the case of Unigram with OR operator, the computed coefficient of determination $R^2 = 0.6847$ shows that there is a strong linear relationship between those two variables. Using the classical linear regression model y = ax + b, we obtain the following results:

```
a = 0.0035, b = 26.629
average execution time = 0.0035 * number of records + 26.629
```

b. Query group 2: 145th query with search string "women clergy"

For all 28 experiments of the 145th query, Table 3.9 presents the computed average execution times (columns AVG) and standard deviations (columns STD) of unigram-based search with AND and OR operators and bigram-based search with AND and OR operators.

We begin by looking at the average execution times. The STD columns will be discussed later in connection with query groups 1 and 3 in section 2.4.4.

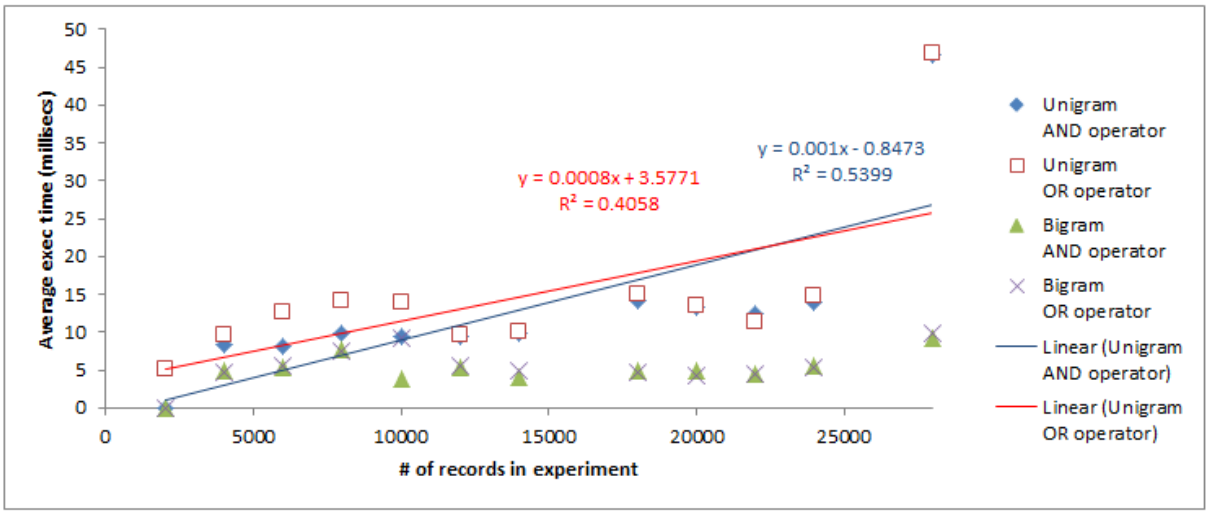
Table 3.9
All experiments for the 145th query

# of records in experiment	Unigram AND operator		Unigram OR operator		Bigram AND operator		Bigram OR operator	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD
2000	n/a	n/a	5.090	3.298	n/a	n/a	n/a	n/a
4000	8.338	2.649	9.679	5.768	4.956	1.676	4.710	2.671
6000	8.226	1.312	12.640	17.126	5.319	1.268	5.469	4.279
8000	9.815	2.479	14.082	13.913	7.661	4.850	7.595	9.841
10000	9.354	1.065	13.854	13.643	3.833	0.839	9.203	24.344
12000	9.345	3.356	9.626	1.977	5.318	1.641	5.521	1.849
14000	9.838	3.694	10.026	2.782	4.125	1.111	4.880	1.628
18000	14.100	2.734	15.026	4.291	4.880	1.896	4.616	1.368
20000	13.281	2.782	13.614	1.591	4.902	1.510	4.170	0.777
22000	12.399	3.595	11.403	1.858	4.472	1.738	4.439	1.028
24000	14.043	3.222	14.787	5.159	5.572	2.089	5.299	1.654
28000	46.660	14.728	46.761	17.232	9.200	2.989	9.900	4.920

The values in the AVG columns are graphically represented in Figure 3.31 with the following conventional colored shapes of markers:

- blue diamond:	unigram, AND operator
- red square:	unigram, OR operator
- olive green triangle:	bigram, AND operator
- X:	bigram, OR operator.

Figure 3.31
All experiments for the 145th query: Average execution times



In Figure 3.31, for 'unigram, AND operator' and 'unigram, OR operator', we observe a common behavior with three characteristics:

- there is an upward trend in average execution times as the number of records in the experiment increases;

- nevertheless, the fluctuation of the averages around the upward trend is more important than in the case of query group 1;

- the figure also suggests that there is a linear relationship between the average execution time and the number of records in experiment.

For 'bigram, AND operator' and 'bigram, OR operator', the behavior is different with the following characteristics:

- there is an horizontal trend in average execution times as the number of records in the experiment increases; in other words, there is very little correlation between the average execution time and the number of records in experiment.

- the average execution times fluctuate slightly around the horizontal trend;

The above characteristics are reflected through the statistical results in Table 3.10, by using the ordinary least squares regression method. As shown in Figure 3.31, the explanatory variable (x) is the number of records in experiments, and the explained variable (y) is the average execution time.

- The coefficients of determination R^2 are very low for 'bigram, AND operator' and 'bigram, OR operator'. Consequently, it is irrelevant to compute the regression coefficients.

- For 'unigram, AND operator' and 'unigram, OR operator', the small values of the regression coefficient (a), 0.00099 and 0.00080, mean that an increase in the number of records from one experiment to the next results in a very small increase in the average execution times.

Table 3.10
All experiments for the 145th query: Linear regression analysis

	R2	a	b	y = ax + b
Unigram, AND operator	0.53988	0.00099	-0.84729	y = 0.00099 x -0.84729
Unigram, OR operator	0.40576	0.00080	3.57709	y = 0.0008 x + 3.57709
Bigram, AND operator	0.27064			
Bigram, OR operator	0.13256			

c. Query group 3: 247th query with search string "air traffic controller"

For all 28 experiments of the 247th query, Table 3.11 presents the computed average execution times (columns AVG) and standard deviations (columns STD) of n-gram-based search with AND and OR operators: unigram, bigram, and trigram.

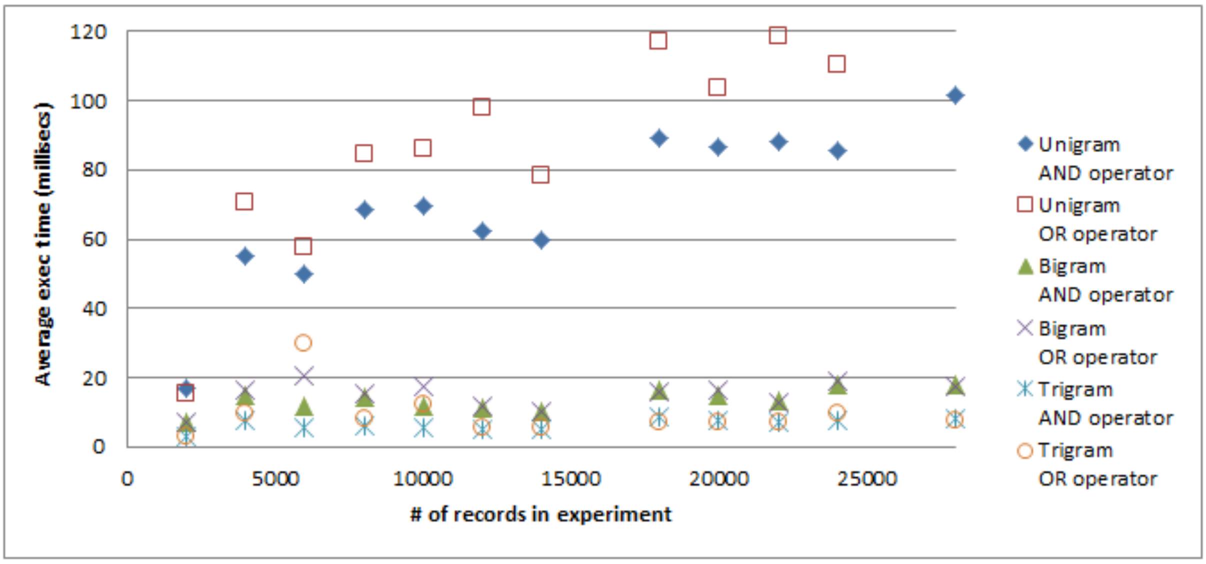
Table 3.11
All experiments for the 247th query

# of records in experiment	Unigram AND operator		Unigram OR operator		Bigram AND operator		Bigram OR operator		Trigram AND operator		Trigram OR operator	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD	AVG	STD	AVG	STD
2000	16.939	3.407	15.412	3.730	7.243	3.159	7.026	2.644	3.176	0.866	3.192	1.166
4000	55.358	23.854	70.389	103.127	14.651	4.794	16.495	8.376	7.711	5.090	9.933	9.019
6000	49.808	13.109	57.717	6.637	11.767	1.548	20.627	49.715	5.696	2.537	29.749	121.225
8000	68.376	7.145	84.371	6.894	14.596	8.796	15.318	16.040	5.921	1.422	7.930	12.873
10000	69.717	7.707	86.273	11.108	11.788	1.707	17.692	22.339	5.437	1.030	12.249	33.429
12000	62.135	5.168	97.786	75.708	11.190	2.856	11.715	2.775	5.004	1.369	5.616	1.390
14000	59.813	3.494	78.131	9.182	9.994	2.153	10.032	2.306	5.103	3.346	5.587	1.648
18000	89.034	7.336	116.810	10.620	16.464	4.406	15.960	1.996	8.896	4.423	7.303	1.501
20000	86.416	33.383	103.459	13.411	15.005	2.829	16.582	5.854	7.659	1.447	7.297	1.747
22000	88.101	15.389	118.845	38.855	13.291	1.852	13.045	3.284	7.203	4.464	6.961	2.548
24000	85.551	7.223	110.133	12.542	17.975	4.352	19.228	17.749	7.668	1.613	9.775	9.649
28000	101.742	8.976	120.879	9.353	17.823	2.631	17.463	3.585	8.223	2.709	7.655	1.479

Let's begin by analyzing the average execution times which are visualized in Figure 3.32 with the following conventional colored shapes of markers:

- blue diamond:	unigram, AND operator
- red square:	unigram, OR operator
- olive green triangle:	bigram, AND operator
- X:	bigram, OR operator
- light blue asterisk:	trigram, AND operator
- orange circle:	trigram, OR operator.

Figure 3.32
All experiments for the 247th query: Average execution times



In Figure 3.32, for 'unigram, AND operator', 'unigram, OR operator', 'bigram, AND operator', and 'trigram, AND operator', we observe a common behavior with two characteristics:

- there is an upward trend in average execution times as the number of records in the experiment increases;

- the figure also suggests a linear relationship between the average execution time and the number of records in experiment.

For 'bigram, OR operator', and 'trigram, OR operator', the behavior is different with the following characteristics:

- there is an horizontal trend in average execution times as the number of records in the experiment increases; in other words, there is very little correlation between the average execution time and the number of records in experiment;

- the average execution times fluctuate slightly around the horizontal trend.

The above characteristics are reflected through the statistical results in Table 3.12, computed by the ordinary least squares regression method. As shown in Figure 3.32, the explanatory variable (x) is the number of records in experiments, and the explained variable (y) is the average execution time.

- The coefficients of determination R^2 are very low for 'bigram, OR operator', and 'trigram, OR operator'. Consequently, it is irrelevant to compute the regression coefficients.

- For 'unigram, AND operator', 'unigram, OR operator', 'bigram, AND operator', and 'trigram, AND operator', the small values of the regression coefficient (a), 0.00244, 0.00314, 0.00027, and 0.00013, mean that an increase in the number of records from one experiment to the next results in a very small increase in the average execution times.

- Another thing we notice is that the regression coefficient (a) for 'unigram, AND operator' (0.00244), and 'unigram, OR operator' (0.00314) are higher than that for bigram, AND operator' (0.00027), and 'trigram, AND operator' (0.00013). That difference could be explained by taking a close look at the way the search query is executed in each case:

    - For 'unigram, AND operator' and 'unigram, OR operator', the execution starts with three individual searches: 'air', 'traffic', and 'controller'. Then, the search results are combined by the operators AND, OR, giving 0.00244 and 0.00314, respectively.

    - For bigram, AND operator', the execution starts with two individual searches 'air:traffic', and 'traffic:controller'. That means a smaller average execution time, 0.00027.

    - For 'trigram, AND operator', the execution starts with a single search 'air:traffic:controller'. That means a smaller average execution time, 0.00013.

Table 3.12
All experiments for the 247th query: Linear regression analysis

	R2	a	b	y = ax + b
Unigram, AND operator	0.79274	0.00244	35.30640	y = 0.00244 x + 35.3064
Unigram, OR operator	0.74211	0.00314	44.42139	y = 0.00314 x + 44.42139
Bigram, AND operator	0.47910	0.00027	9.75131	y = 0.00027 x + 9.75131
Bigram, OR operator	0.08147			
Trigram, AND operator	0.44409	0.00013	4.61858	y = 0.00013 x + 4.61858
Trigram, OR operator	0.05673			

d. Examining the standard deviations of query execution times

So far we did not yet discussed the standard deviations reported in previous Table 3.8, Table 3.9, and Table 3.11.

According to the experimentation plan in section a (Table 3.8, Table 3.9, and Table 3.11), a standard deviation of execution times is computed after every 25 runs of each query in a given experiment. For the purpose of our discussion in this section, Table 3.13 displays the standard deviations computed from experiments nos 9 and 10.

Table 3.13 can be analyzed horizontally from left to right, and vertically from top to bottom.

Horizontal analysis of standard deviation

Let's look at query no.1 in experiment no.9 in Table 3.13. Its search string is `'International Organized Crime'`.

a) **First**, the search query module processes 'unigram, AND operator'. The very first run out of 25 is the most time consuming. Indeed, since the cache is initially empty, Cassandra has to access the index to search the column families containing the documents' IDs associated with unigrams '`International`', '`Organized`', and '`Crime`', and progressively fills the cache with the column families found. At the same time, Cassandra returns these column families to the 'unigram, AND operator' routine of the search query module which then generates the search results by ANDing the three unigrams.

The remaining 24 runs are much less time consuming, since the column families can be readily found in the cache. Hence, the search query module will generate the results more rapidly.

Consequently, the sample of 25 execution times is characterized by a large value of the first observed execution time, and 24 much smaller remaining values. Hence, the standard deviation is big (37.4415) for 'unigram, AND operator' of query no.1.

b) **Second**, the search query module processes 'bigram, AND operator'. At this moment, the cache contains only unigram column families. Therefore, Cassandra still has to access the index to search the column families containing the documents' IDs associated with bigrams '`International:Organized`', and '`Organized:Crime`'., and progressively fills the cache with the column families found. At the same time, Cassandra returns these column families to the 'bigram, AND operator' routine of the search query module which then generates the search results by ANDing the two bigrams.

In our particular set of 10'000 documents' data, there is no such result for all 25 runs; hence, the symbol `n/a` for 'bigram, AND operator' in Table 3.13. Although it takes time to access the index, and fill the cache, the average and the standard deviation are not computed, since the result is empty

c) **Third**, the search query module processes 'trigram, AND operator'. Analogously, Cassandra progressively fills the cache with the trigram '`International:Organized:Crime`' column family. The result of ANDing the trigram with itself is empty; hence, the symbol `n/a` for 'trigram, AND operator' in Table 3.13.

d) **Fourth**, the search query module processes 'unigram, OR operator'. Since the cache is already filled in step a), Cassandra doesn't need to access the index. Instead, only the cache will be searched by Cassandra which returns the found column families to the 'unigram, OR operator' routine of the search query module. The routine then generates the search results by ORing the three unigrams.

Thanks to the cache, there is no big difference between the execution time of the first run and those of the 24 remaining runs. Hence, the standard deviation (8.1628) for 'unigram, OR operator' is much less than that for 'unigram, AND operator' (37.4415) in step a).

e) **Fifth**, similary, for 'bigram, OR operator', the standard deviation of execution times (1.9908) is also much less than that for 'unigram, AND operator' (37.4415) in step a).

f) **Sixth**, the symbol `n/a` appears for 'trigram, OR operator', since there is no result found for 'trigram, OR operator'.

Vertical analysis of standard deviation

Now, let's look at the column 'unigram, AND operator' in Table 3.13. The standard deviation of execution time (37.4415) is large for query no.1, as explained in step a), section a.

For subsequent queries no.2 thru 250, Cassandra needs to access the index only if the search terms do not exist in the cache yet. That is why the standard deviations of execution time for all queries but the first are relatively lower than that of the first query.

Table 3.13
Experiments nos 9 and 10: Standard deviations of query execution times

Experiment no.	# of scientific papers	Query no.	Unigram AND	Bigram AND	Trigram AND	Unigram OR	Bigram OR	Trigram OR
9	9000	1	37.4415	n/a	n/a	8.1628	1.9908	n/a
		2	n/a	n/a	n/a	4.2610	n/a	n/a
		3	n/a	n/a	n/a	1.9330	n/a	n/a
		4	n/a	n/a	n/a	0.3749	n/a	n/a
		...	...	...	...	...	...	...
		145	4.9259	2.3275	n/a	0.3767	0.0546	n/a
		...	...	...	...	...	...	...
		247	7.2931	1.1811	6.3654	2.1916	1.6079	1.1618
		...	...	...	...	...	...	...
		248	n/a	n/a	n/a	3.0153	0.9855	n/a
		249	2.1183	n/a	n/a	2.1449	n/a	n/a
		250	n/a	n/a	n/a	4.4354	0.3977	n/a
10	10000	1	39.1168	n/a	n/a	8.2990	1.4130	n/a
		2	n/a	n/a	n/a	0.3749	n/a	n/a
		3	n/a	n/a	n/a	0.2972	n/a	n/a
		4	n/a	n/a	n/a	0.3749	n/a	n/a
		...	...	...	...	...	...	...
		145	3.7151	0.4447	n/a	0.2146	0.0769	n/a
		...	...	...	...	...	...	...
		247	5.0682	0.2340	4.6105	3.3891	1.5057	0.3053
		...	...	...	...	...	...	...
		248	n/a	n/a	n/a	1.0618	0.1324	n/a
		249	3.1591	n/a	n/a	1.8359	n/a	n/a
		250	n/a	n/a	n/a	3.3599	0.3767	n/a

e. Search performance comparison between multi and single node cluster

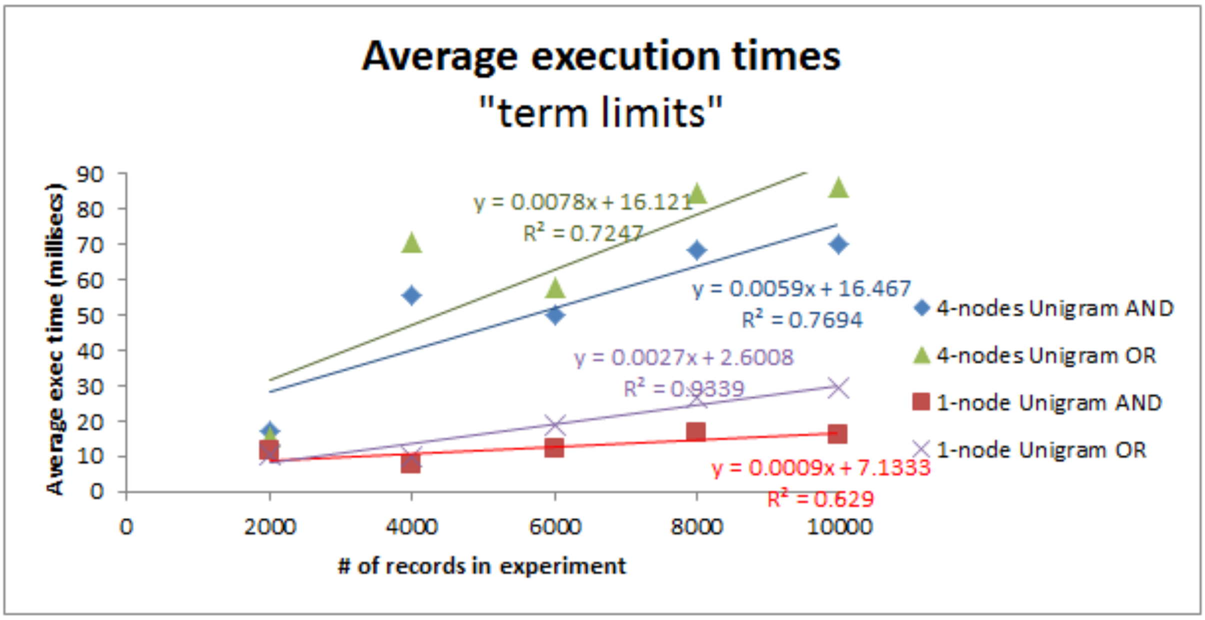
The results in Table 3.2 show that it takes more time to write on multi-node cluster than on single node cluster. According to the theory on read operation presented in section 1.4.5, it also takes more time to read on multi-node cluster.

The results of experiments with unigrams, bigrams, and trigrams confirm the above theoretical conclusion. The graphical representation in Figure 3.33 illustrates the case of unigram in the 249th search query "term limits" with the following conventional colored shapes of markers:

- blue diamond:	Unigram AND operator on 4-node cluster
- olive green triangle:	Unigram OR operator on 4-node cluster
- red square:	Unigram AND operator on single node cluster
- purple X:	Unigram OR operator on single node cluster.

The blue and olive green curves corresponding to the 4-node cluster are above the red and purple curves of the single node cluster.

Figure 3.33
All experiments for the 249th query



We observe that four curves follow the same behavior. More precisely, Figure 3.30 suggests that there might be a linear relationship between the number of records and the average execution time. The computed coefficient of determinations R^2 show that there is a strong linear relationship between those two variables. Using the classical linear regression model y = ax + b, we obtain the following results:

- Unigram AND operator on 4-node cluster:

```
average execution time = 0.0059 * number of records + 16.467
```

- Unigram OR operator on 4-node cluster:

```
average execution time = 0.0078 * number of records + 16.121
```

- Unigram AND operator on sing node cluster:

```
average execution time = 0.0009 * number of records + 7.1333
```

- Unigram OR operator on sing node cluster:

```
average execution time = 0.0027 * number of records + 2.6008
```

Furthermore, read performance on single node cluster is always faster than on multi-node cluster. Indeed, the regression coefficient (a) for '4-node, unigram AND' (0.0059), and '4-node, unigram OR' (0.0078) are higher than that for '1-node, unigram AND' (0.0009), and '1-

node, unigram OR' (0.0027). Those numbers show also that an increase in the number of records from one experiment to the next has little effect on the average execution times.

3.3 Experimentation on Solr single node cluster

3.3.1 Understanding the search query module in Java

The search strings used in queries are stored in an XML file as 250 XML `title` elements, one for each query. They are identical to those already used in the previous experimentation with Cassandra.

Figure 3.34 shows the Java code that forms the queries based on the `title` element, as explained below:

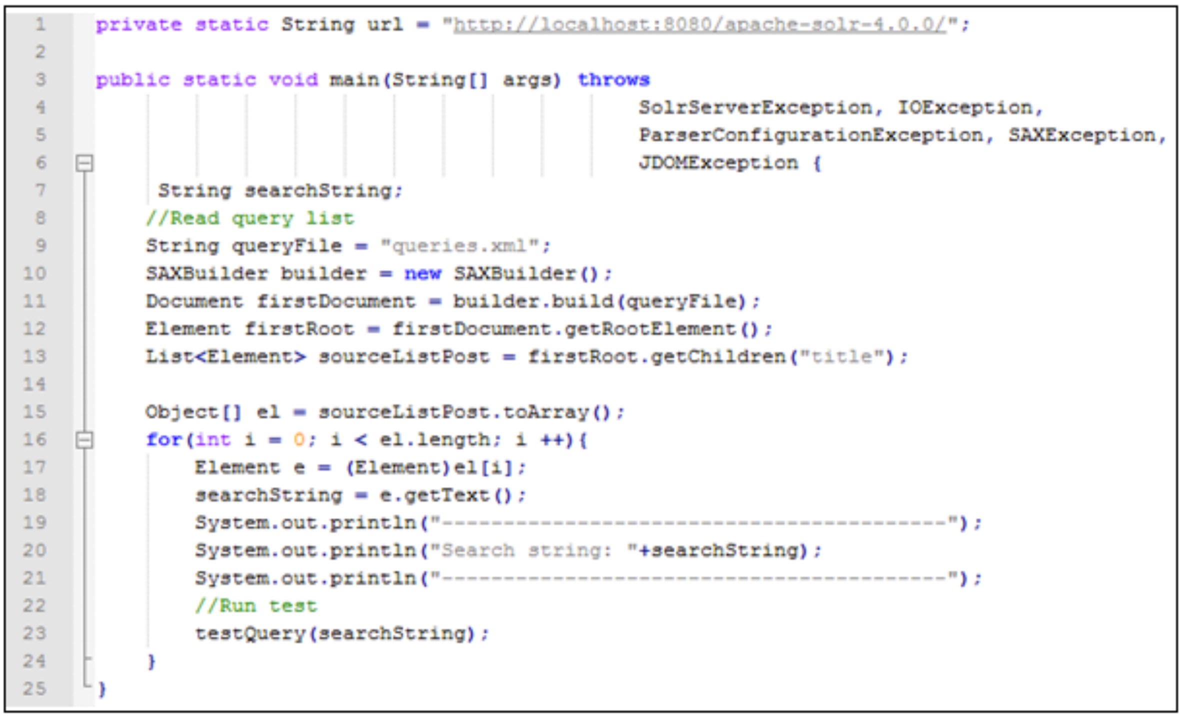
- line 9: The 250 queries are stored in the local file `queries.xml`.

- line 16: The `for` statement specifies that the loop will be executed 250 times (`el.length`), one for each query.

- line 18: A `searchString` corresponding to the ith query is extracted from `queries.xml`.

- line 23: The `testQuery` module is invoked to process the ith query with `searchString` as input argument.

Figure 3.34
Solr: Java code to form the queries



```
1   private static String url = "http://localhost:8080/apache-solr-4.0.0/";
2   
3   public static void main(String[] args) throws
4                                           SolrServerException, IOException,
5                                           ParserConfigurationException, SAXException,
6                                           JDOMException {
7       String searchString;
8       //Read query list
9       String queryFile = "queries.xml";
10      SAXBuilder builder = new SAXBuilder();
11      Document firstDocument = builder.build(queryFile);
12      Element firstRoot = firstDocument.getRootElement();
13      List<Element> sourceListPost = firstRoot.getChildren("title");
14  
15      Object[] el = sourceListPost.toArray();
16      for(int i = 0; i < el.length; i ++){
17          Element e = (Element)el[i];
18          searchString = e.getText();
19          System.out.println("-----------------------------------------");
20          System.out.println("Search string: "+searchString);
21          System.out.println("-----------------------------------------");
22          //Run test
23          testQuery(searchString);
24      }
25  }
```

This module is now explained in detail and illustrated by Figure 3.35.

There are two ways to connect to a Solr server:

- The simplest and safest way is to connect to a Solr server by using HTTP;

- The second method is to connect to Solr core by using the concept of 'Embedded Solr'. This is not recommended in production environment, because it is "less flexible, harder to support, not as well tested, and should be reserved for special circumstances" [apache wiki].

The first connection method was selected in our application.

Figure 3.35 shows the Java `testQuery` module that executes a query with `searchString` as search argument:

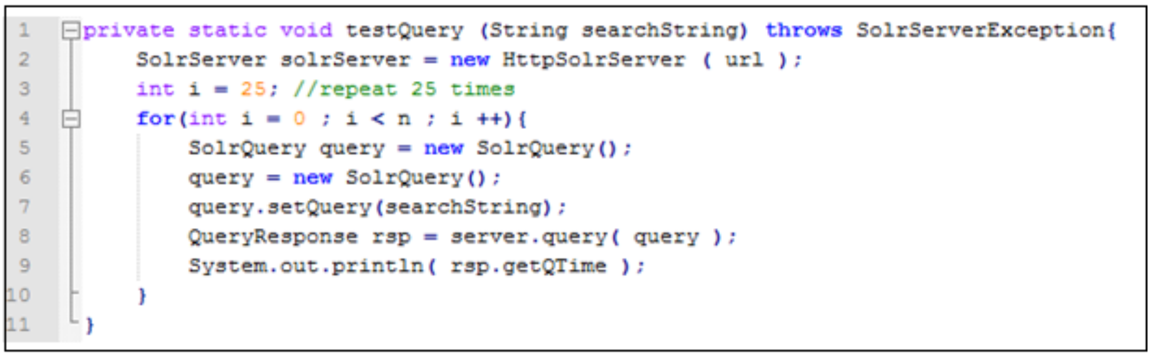
- line 2: Connect to Solr using standard HTTP interfaces.

- line 4: The `for` statement specifies that a given query will be executed 25 times.

- line 8: Run the query.

- line 9: Get statistical information, such as: query execution time.

Figure 3.35
testQuery module: Java code to execute a search query



```
 1  private static void testQuery (String searchString) throws SolrServerException{
 2      SolrServer solrServer = new HttpSolrServer ( url );
 3      int i = 25; //repeat 25 times
 4      for(int i = 0 ; i < n ; i ++){
 5          SolrQuery query = new SolrQuery();
 6          query = new SolrQuery();
 7          query.setQuery(searchString);
 8          QueryResponse rsp = server.query( query );
 9          System.out.println( rsp.getQTime );
10      }
11  }
```

3.3.2 Experimentation plan

In order to compare the performance of Cassandra to Solr, the experimentation plan designed for Cassandra will be applied to Solr.

Let's recall that each experiment is assigned a predefined number of scientific papers. For example, Experiment no.1 has 1'000 scientific papers, Experiment no.2 has 2'000 scientific papers, Experiment no.10 has 10'000 scientific papers.

10 experiments will be carried out. As illustrated by the worksheets in Table 3.14, Table 3.15, and Table 3.16, for Experiments nos 1, 2, and 10, respectively; each experiment will process sequentially 250 predefined queries. Each query will be run

- 25 times with AND operator,
- 25 times with OR operator,

Observed execution times are reported into the shaded areas of the tables.

Table 3.14
Plan of Experiment no.1

Experiment no.	# of scientific papers	Query no.	Run no.	AND operator	Run no.	OR operator
1	1000	1	1		26	
	...		...		...	
	1000		25		50	
	1000	2	1		26	
	...		...		...	
	1000		25		50	
	...	...	...		...	
	1000	250	1		26	
	...		...		...	
	1000		25		50	

Table 3.15
Plan of Experiment no.2

Experiment no.	# of scientific papers	Query no.	Run no.	AND operator	Run no.	OR operator
2	2000	1	1		26	
	...		...		...	
	2000		25		50	
	2000	2	1		26	
	...		...		...	
	2000		25		50	
	...	...	...		...	
	2000	250	1		26	
	...		...		...	
	2000		25		50	

Table 3.16
Plan of Experiment no.10

Experiment no.	# of scientific papers	Query no.	Run no.	AND operator	Run no.	OR operator
10	10000 ... 10000	1	1 ... 25		26 ... 50	
	10000 ... 10000	2	1 ... 25		26 ... 50	
	...	...	...		...	
	10000 ... 10000	250	1 ... 25		26 ... 50	

3.4 Performance comparison between Cassandra and Solr on single node cluster

Looking back at the worksheets in Table 3.14, Table 3.15, and Table 3.16, we notice that in Solr the n-gram feature is present, but without unigram, bigram, and trigram separately implemented. Therefore, in order to make comparable the observed performance (query execution times) under Cassandra and Solr, the execution times for unigram, bigram and trigram in Cassandra must be aggregated for AND, and OR operators, respectively, as illustrated by the two rounded rectangles in Table 3.17 for Query no.1 in Experiment no.1.

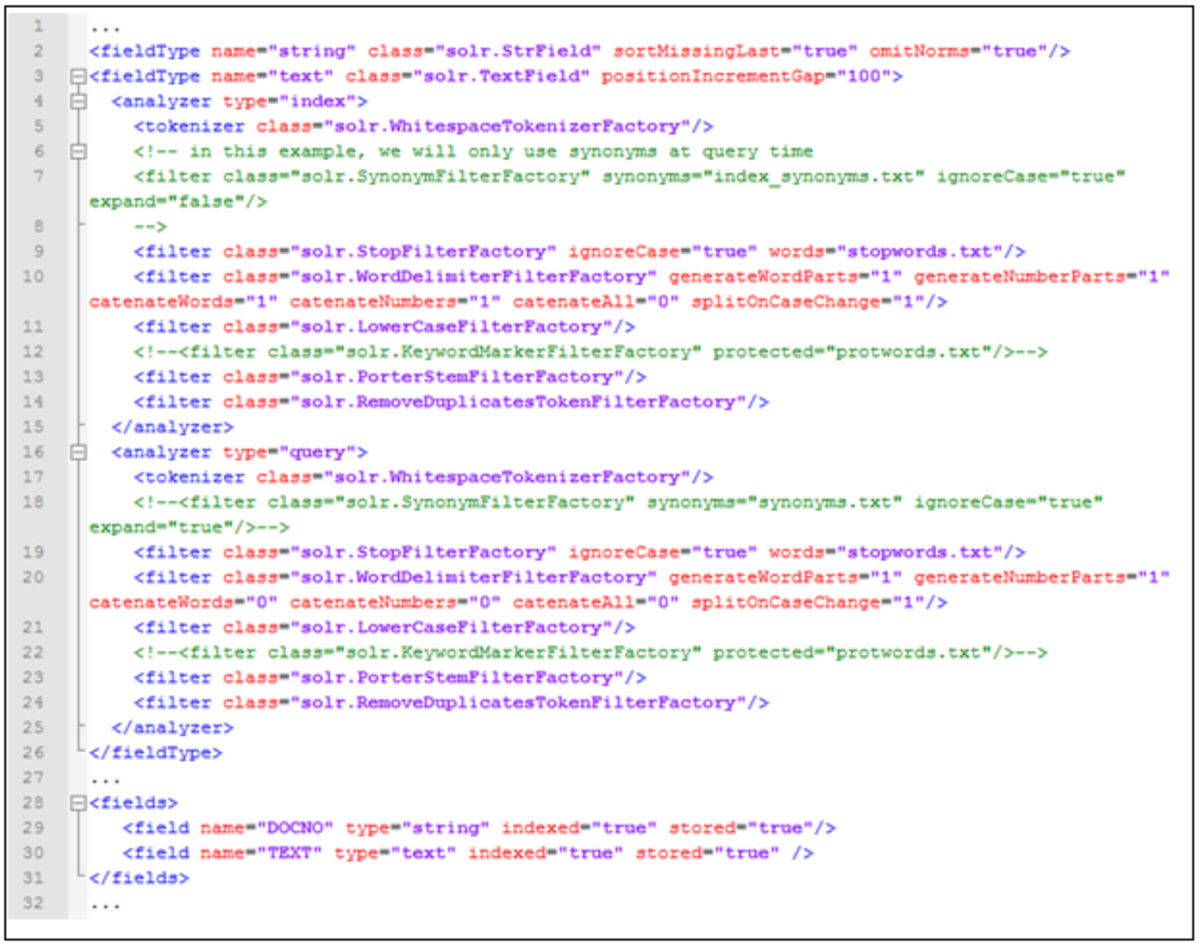
Table 3.17
Aggregation of execution times for AND, OR operators

Experiment no.	# of scientific papers	Query no.	Run no.	Unigram AND	Run no.	Bigram AND	Run no.	Trigram AND	Run no.	Unigram OR	Run no.	Bigram OR	Run no.	Trigram OR
1	1000 ... 1000	1	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	1000 ... 1000	2	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	
	...	...	...		...		...		...		...		...	
	1000 ... 1000	250	1 ... 25		26 ... 50		51 ... 75		76 ... 100		101 ... 125		126 ... 150	

In Section 1.5 of Chapter 1, we emphasized the importance of two files in Solr: `schema.xml` and `solrconfig.xml`. A look at the `<filter>` elements in `schema.xml` (Figure 3.36) suggests that Solr performs the same steps as Cassandra:

removing blank spaces, removing html code, stemming, removing a particular character and replacing it with another one.

Figure 3.36
Excerpt of schema.xml



```xml
...
<fieldType name="string" class="solr.StrField" sortMissingLast="true" omitNorms="true"/>
<fieldType name="text" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt" ignoreCase="true"
expand="false"/>
    -->
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1" generateNumberParts="1"
catenateWords="1" catenateNumbers="1" catenateAll="0" splitOnCaseChange="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <!--<filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>-->
    <filter class="solr.PorterStemFilterFactory"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <!--<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true"
expand="true"/>-->
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1" generateNumberParts="1"
catenateWords="0" catenateNumbers="0" catenateAll="0" splitOnCaseChange="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <!--<filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>-->
    <filter class="solr.PorterStemFilterFactory"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>
...
<fields>
   <field name="DOCNO" type="string" indexed="true" stored="true"/>
   <field name="TEXT" type="text" indexed="true" stored="true" />
</fields>
...
```

Our comparative study is based on the performance of queries nos. 249th, 145th, and 247th selected from the query groups 1, 2, and 3, respectively, even though Solr is not concerned about unigram, bigram, and trigram.

3.4.1 Query group 1: 249th query with search string "term limits"

For all 10 experiments of the 249th query, Table 3.18 presents the average execution times (columns AVG) and standard deviations (columns STD) for AND and OR operators on Solr and Cassandra.

Table 3.18
All experiments for the 249th query

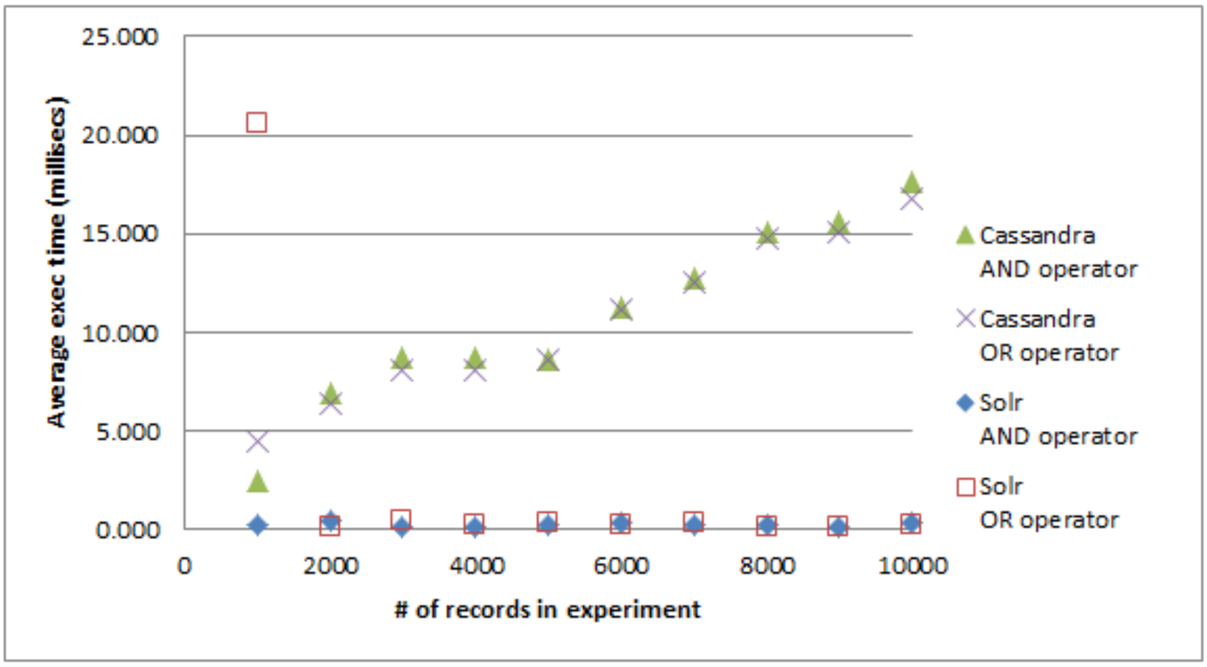
# of records in experiment	Solr				Cassandra			
	AND operator		OR operator		AND operator		OR operator	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD
1000	0.240	0.427	20.560	99.499	2.476	0.559	4.480	2.637
2000	0.440	0.852	0.200	0.400	6.959	6.002	6.396	5.460
3000	0.120	0.325	0.520	0.574	8.759	4.551	8.062	2.724
4000	0.120	0.325	0.280	0.531	8.685	4.526	8.095	3.282
5000	0.280	0.449	0.320	0.466	8.621	2.645	8.662	2.669
6000	0.360	0.480	0.240	0.427	11.312	2.303	11.117	3.484
7000	0.280	0.531	0.320	0.466	12.801	1.822	12.578	1.512
8000	0.280	0.449	0.200	0.400	15.104	3.606	14.802	2.125
9000	0.200	0.400	0.200	0.490	15.578	2.118	15.131	2.145
10000	0.320	0.466	0.240	0.427	17.611	3.159	16.747	1.836

We begin by looking at the average execution times.

The values in the AVG columns are graphically represented in Figure 3.37 with the following conventional colored shapes of markers:

- blue diamond:	Solr, AND operator
- red square:	Solr, OR operator
- olive green triangle:	Cassandra, AND operator
- X:	Cassandra, OR operator.

Figure 3.37
All experiments for the 249th query: Average execution times



Overall, the performance obtained with Cassandra is lower than that with Solr regardless of the operators AND, and OR.

For 'Cassandra, AND operator', and 'Cassandra, OR operator', we observe a common behavior with two characteristics:

- there is an upward trend in average execution times as the number of records in the experiment increases;

- the figure also suggests a linear relationship between the average execution times and the number of records in experiments.

'Solr, AND operator', and 'Solr, OR operator' show another common behavior with three characteristics:

- there is an horizontal trend in average execution times as the number of records in the experiment increases; in other words, there is very little correlation between the average execution time and the number of records in experiment;

- the fluctuation of average execution times around the horizontal trend looks insignificant;

- for the first experiment (1'000 records) (Figure 3.37), the average execution time of the 249th query with OR operator in Solr takes on value that could be considered as outliers (20.560 millisecs).

The two different behaviors of Cassandra and Solr described above are formalized through the statistical results in Table 3.19, by using the ordinary least squares regression method. The explanatory variable (x) is the number of records in experiments, and the explained variable (y) is the average execution time.

- As expected, the coefficients of determination R^2 are very low for 'Solr, AND operator', and 'Solr, OR operator'. Consequently, it is irrelevant to compute the regression coefficients.

- For 'Cassandra, AND operator', and 'Cassandra, OR operator', the small positive values of the regression coefficient (a), 0.00147 and 0.00134, mean that an increase in the number of records from one experiment to the next has little effect on the increase of the average execution times.

Table 3.19
All experiments for the 249th query: Linear regression analysis

	R2	a	b	y = ax + b
Solr, AND operator	0.00537			
Solr, OR operator	0.27843			
Cassandra, AND operator	0.94698	0.00147	2.67999	y = 0.00147x + 2.67999
Cassandra, OR operator	0.97663	0.00134	3.23536	y = 0.00134x + 3.23536

3.4.2 Query group 2: 145th query with search string "women clergy"

For all 10 experiments of the 145th query, Table 3.20 presents the average execution times (columns AVG) and standard deviations (columns STD) for AND and OR operators on Solr and Cassandra.

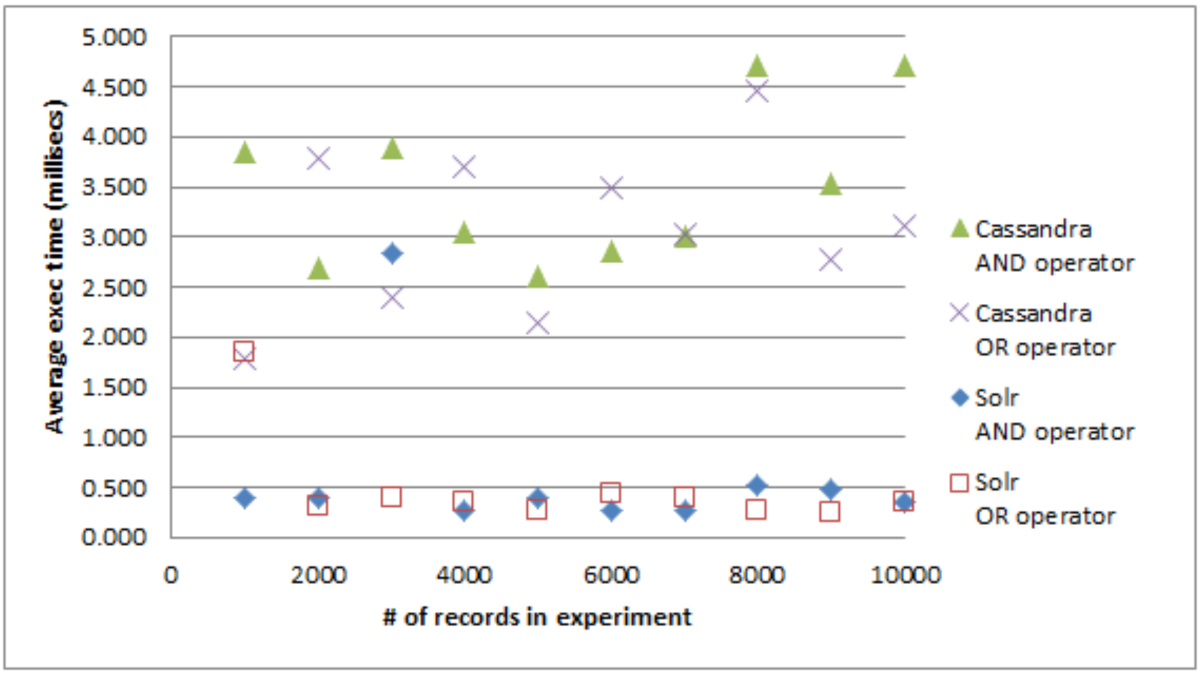
Table 3.20
All experiments for the 145th query

# of records in experiment	**Solr**				**Cassandra**			
	AND operator		OR operator		AND operator		OR operator	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD
1000	0.400	0.490	1.840	7.598	3.858	2.553	1.789	0.756
2000	0.400	0.490	0.320	0.466	2.699	0.352	3.790	3.533
3000	2.840	12.085	0.400	0.490	3.901	2.597	2.390	1.184
4000	0.280	0.449	0.360	0.480	3.050	2.152	3.694	2.365
5000	0.400	0.849	0.280	0.449	2.601	2.490	2.141	1.510
6000	0.280	0.449	0.440	0.753	2.860	3.850	3.486	2.740
7000	0.280	0.449	0.400	0.849	3.010	2.406	3.033	2.348
8000	0.520	0.755	0.280	0.449	4.701	4.546	4.452	3.274
9000	0.480	0.700	0.240	0.650	3.535	4.435	2.769	2.122
10000	0.360	0.480	0.360	0.480	4.703	4.227	3.119	2.012

We begin by looking at the average execution times. The values in the AVG columns are graphically represented in Figure 3.38 with the following conventional colored shapes of markers:

- blue diamond:	Solr, AND operator
- red square:	Solr, OR operator
- olive green triangle:	Cassandra, AND operator
- X:	Cassandra, OR operator.

Figure 3.38
All experiments for the 145th query: Average execution times



Overall, the performance obtained with Cassandra is again lower than that with Solr regardless of the operators AND, and OR. In other words, it takes more time on average to run the queries on Cassandra than that on Solr.

For 'Cassandra, AND operator', and 'Cassandra, OR operator', we observe a common behavior with two characteristics:

- there is an upward trend in average execution times as the number of records in the experiment increases;

- nevertheless, the fluctuation of the averages around the trend is more important than in the case of query group 1. Consequently, the linear relationship between the two variables is less obvious than in query group 1.

'Solr, AND operator', and 'Solr, OR operator' show another common behavior with two characteristics:

- there is an horizontal trend in average execution times as the number of records in the experiment increases; in other words, there is very little correlation between these two variables.

- the average execution times fluctuate slightly around the horizontal trend;

3.4.3 Query group 3: 247th query with search string "air traffic controller"

For all 10 experiments of the 145th query, Table 3.21 presents the average execution times (columns AVG) and standard deviations (columns STD) for AND and OR operators on Solr and Cassandra.

Table 3.21
All experiments for the 247th query

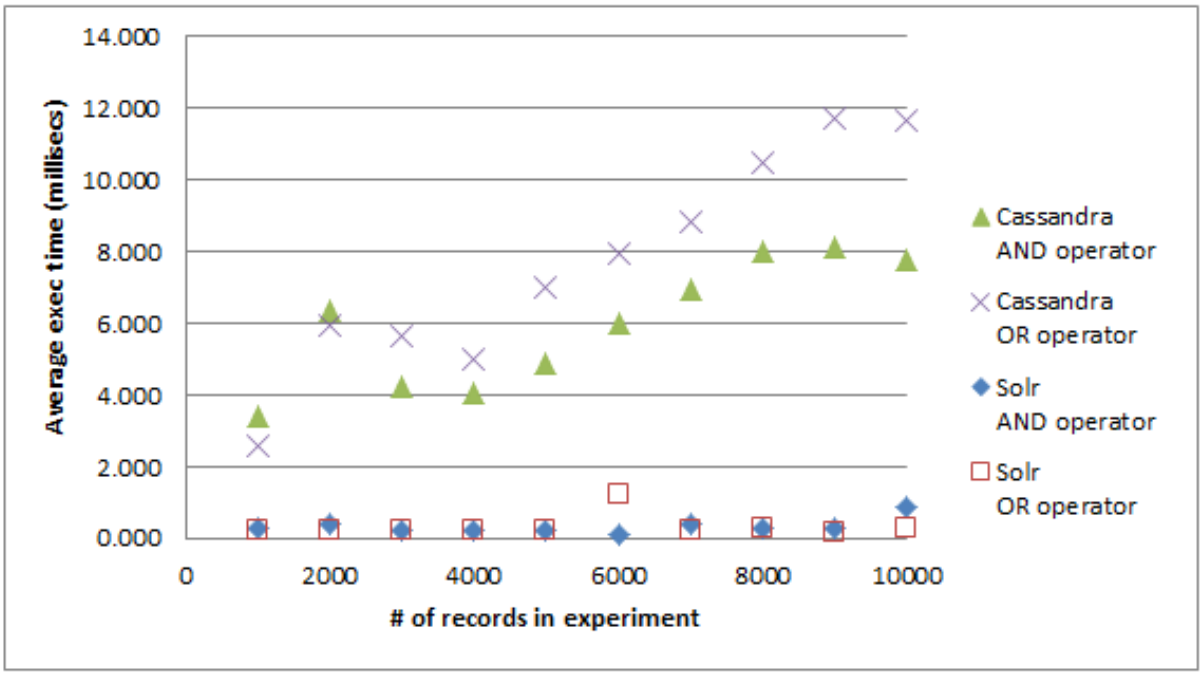
# of records in experiment	Solr				Cassandra			
	AND operator		OR operator		AND operator		OR operator	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD
1000	0.280	0.449	0.240	0.512	3.393	2.116	2.593	3.888
2000	0.400	0.849	0.240	0.427	6.362	7.531	5.930	5.822
3000	0.240	0.512	0.200	0.400	4.239	3.739	5.669	5.890
4000	0.240	0.512	0.240	0.427	4.077	3.471	4.996	4.489
5000	0.200	0.400	0.200	0.400	4.911	4.904	7.006	4.792
6000	0.080	0.271	1.200	4.079	6.010	5.890	7.936	8.017
7000	0.400	1.575	0.200	0.400	6.966	8.402	8.847	9.348
8000	0.280	0.449	0.280	0.449	8.003	7.539	10.492	11.405
9000	0.280	0.449	0.160	0.367	8.127	7.961	11.696	10.979
10000	0.880	3.514	0.280	0.449	7.745	7.193	11.654	12.838

We begin by looking at the average execution times.

The values in the AVG columns are graphically represented in Figure 3.39 with the following conventional colored shapes of markers:

- blue diamond:	Solr, AND operator
- red square:	Solr, OR operator
- olive green triangle:	Cassandra, AND operator
- X:	Cassandra, OR operator.

Figure 3.39
All experiments for the 247th query: Average execution times



Overall, the performance obtained with Cassandra is lower than that with Solr, regardless of the operators AND, and OR.

For ‘Cassandra, AND operator’, and ‘Cassandra, OR operator’, we observe a common behavior with two characteristics:

- there is an obvious upward trend in average execution times as the number of records in the experiment increases;

- the figure also suggests a linear relationship between the two variables.

‘Solr, AND operator’, and ‘Solr, OR operator’ show a different common behavior with two characteristics:

- there is an horizontal trend in average execution times as the number of records in the experiment increases; in other words, there is very little correlation between the two variables;

- the fluctuation of average execution times around the horizontal trend looks insignificant;

The two different behaviors of Cassandra and Solr described above are formalized through the statistical results in Table 3.22, by using the ordinary least squares regression method. The explanatory variable (x) is the number of records in experiments, and the explained variable (y) is the average execution time.

- As expected, the coefficients of determination R^2 are very low for ‘Solr, AND operator’, and ‘Solr, OR operator’. Consequently, it is irrelevant to compute the regression coefficients.

- For ‘Cassandra, AND operator’, and ‘Cassandra, OR operator’, the small positive values of the regression coefficient (a), 0.00049 and 0.00096, mean that an increase in the number of records from one experiment to the next has little effect on the increase of the average execution times.

Table 3.22
All experiments for the 249th query: Linear regression analysis

	R2	a	b	y = ax + b
Solr, AND operator	0.19125			
Solr, OR operator	0.00408			
Cassandra, AND operator	0.70908	0.00049	3.31283	y = 0.00049x + 3.31283
Cassandra, OR operator	0.92965	0.00096	2.39776	y = 0.00096x + 2.39776

3.5 Conclusion

3.5.1 The findings

Given the specific properties of the two environments set up for Cassandra and Solr in this research, and under the experimentation plans designed for both systems, our main findings about their performance are summarized in this section.

In the Cassandra environment which is the focus of this research, the analysis of experimental performance data shows the following results:

- Unigram AND, Unigram OR: strong linear relationship between the number of scientific papers in experiment (explanatory variable) and the average query execution time (explained variable); upward linear trend between these two variables;

- Bigram AND, and Trigram AND: weak linear relationship; weak upward trend;

- Bigram OR, and Trigram OR: no linear relationship;

- Either with unigram, bigram or trigram, write and read performance on a multi-node cluster are slightly lower than the performance on a single node cluster.

An interesting finding is that there is no linear relationship between the number of scientific papers and the average query execution time when the experimentation uses 10'000 scientific papers. In constrast, if this number is increased to 28’000, there exists a linear relationship between these explained and explanatory variables for Bigram AND.

The comparative performance study of Cassandra and Solr on single node cluster results in the following findings:

- Overall, the performance obtained with Cassandra is lower than that with Solr, for both operators AND, and OR.

- In contrast to Cassandra, quey execution time in Solr always shows practically no linear relationship with the number of scientific papers in the experiments.

3.5.2 Next step

Searching is an operation requiring high performance. Therefore, the next step could consist in exploring solutions to optimize the search mechanism with Cassandra, in particular, through the use of Cassandra's *built-in key* and *row caches*?

Source Code

Listing 0.1 Module testQueries

```
private static void testQueries() throws JDOMException, IOException,
SolrServerException {
    String searchString;
    //Read queries
    String queryFile = "queries.xml";
    SAXBuilder builder = new SAXBuilder();
    Document firstDocument = builder.build(queryFile);
    Element firstRoot = firstDocument.getRootElement();
    List<Element> sourceListPost = firstRoot.getChildren("title");
    Object[] el = sourceListPost.toArray();
    for(int i = 0; i < el.length; i ++){
        Element e = (Element)el[i];
        searchString = e.getText();
        System.out.println("-----------------------------------------");
        System.out.println("Search string: "+searchString);
        System.out.println("-----------------------------------------");
        //Run test
        testQuery(searchString);
    }
}
```

Listing 0.2 Module testQuery

```
private static void testQuery(String searchString) throws
SolrServerException {
    SolrServer embeddedServer = new HttpSolrServer(url);
    int n = 25; //repeat 25 times
    for (int i = 0; i < n ; i ++){
        SolrQuery query = new SolrQuery();
        query = new SolrQuery();
        query.setQuery(searchString);
        QueryResponse rsp2 = embeddedServer.query(query);
        System.out.println(rsp2.getQTime());
    }
}
```

Listing 0.3 Module endElement

```
public void endElement(String uri, String localName, String qName) throws
SAXException {
    if(qName.equalsIgnoreCase("DOC")) {
        //add it to the list
```

```
            myDocuments.add(tempDoc);
        }else if (qName.equalsIgnoreCase("DOCNO")) {
            tempDoc.setDocNo(tempVal);
        }else if (qName.equalsIgnoreCase("FILEID")) {
            tempDoc.setFileID(tempVal);
        }else if (qName.equalsIgnoreCase("FIRST")) {
            tempDoc.setFirst(tempVal);
        }else if (qName.equalsIgnoreCase("SECOND")) {
            tempDoc.setSecond(tempVal);
        }else if (qName.equalsIgnoreCase("HEAD")) {
            tempDoc.setHead(tempVal);
        }else if (qName.equalsIgnoreCase("DATELINE")) {
            tempDoc.setDateline(tempVal);
        }else if (qName.equalsIgnoreCase("TEXT")) {
            try {
                TreeMap<String, Integer> Text = new TreeMap<String, Integer>();
                String accumulatorString =
    removeStopWordsAndStem(accumulator.toString());
                Scanner scanner = new Scanner(accumulatorString);
                scanner.useDelimiter("[^\\p{Alpha}\\p{Digit}]+");
                String word1, word2, word3;
                if(scanner.hasNext()){
                    word3 = scanner.next().toLowerCase();
                    if(Text.containsKey((String)(word3))){
                        Integer freq = Text.get((String)(word3));
                        Text.put(((String)(word3)), freq + 1);
                    }
                    if (!Text.containsKey((String)(word3))){
                        Text.put(((String)(word3)), 1);
                    }
                    if(scanner.hasNext()){
                        word2 = word3;
                        word3 = scanner.next().toLowerCase();
                        if(Text.containsKey((String)(word3))){
                            Integer freq = Text.get((String)(word3));
                            Text.put(((String)(word3)), freq + 1);
                        }
                        if(Text.containsKey((String)(word2+":"+word3))){
                            Integer freq = Text.get((String)(word2+":"+word3));
                            Text.put(((String)(word2+":"+word3)), freq + 1);
                        }
                        if (!Text.containsKey((String)(word3))){
                            Text.put(((String)(word3)), 1);
                        }
                        if (!Text.containsKey((String)(word2+":"+word3))){
                            Text.put(((String)(word2+":"+word3)), 1);
                        }
                        while(scanner.hasNext()){
                            word1 = word2;
                            word2 = word3;
                            word3 = scanner.next().toLowerCase();
                            if(Text.containsKey((String)(word3))){
                                Integer freq = Text.get((String)(word3));
                                Text.put(((String)(word3)), freq + 1);
                            }
                            if(Text.containsKey((String)(word2+":"+word3))){
                                Integer freq =
    Text.get((String)(word2+":"+word3));
                                Text.put(((String)(word2+":"+word3)), freq +
    1);
                            }

    if(Text.containsKey((String)(word1+":"+word2+":"+word3))){
                                Integer freq =
    Text.get((String)(word1+":"+word2+":"+word3));
                                Text.put(((String)(word1+":"+word2+":"+word3)),
    freq + 1);
                            }
                            if (!Text.containsKey((String)(word3))){
                                Text.put(((String)(word3)), 1);
                            }
                            if (!Text.containsKey((String)(word2+":"+word3))){
                                Text.put(((String)(word2+":"+word3)), 1);
                            }
                            if
    (!Text.containsKey((String)(word1+":"+word2+":"+word3))){
                                Text.put(((String)(word1+":"+word2+":"+word3)),
    1);
                            }
                        }
                    }
                }
                tempDoc.setText(Text);
            } catch (IOException ex) {

    Logger.getLogger(SAXParserNGram.class.getName()).log(Level.SEVERE, null,
    ex);
            }
        }
```

Listing 0.4 Module testANDUnigram

```
public static  void testANDUnigram(String searchString,String fileName)
throws FileNotFoundException, TTransportException, TException,
InvalidRequestException, IOException, UnavailableException,
TimedOutException{
    connector = new Connector();
    client = connector.connect();
    Long clockStart;
    Long clockFinish;
    Set<String> postingList;

    SAXParserNGram ngram = new SAXParserNGram();
    searchString = ngram.removeStopWordsAndStem(searchString);
    //Run 25 times
    int i, n = 25;
    List<ByteBuffer> setUnigram = new ArrayList<>();
    System.out.println("************************");
    System.out.println("** intersection unigram**");
    System.out.println("************************");
    setUnigram = splitPhraseUnigram(searchString,fileName);
    for (i = 0; i < n; i++)
    {
        clockStart = System.nanoTime();
        postingList = intersection(setUnigram, Constants.CFamily);
        clockFinish = System.nanoTime();
        if (postingList.size()==0)
            System.out.println("n/a");
        else
            System.out.println((double)((clockFinish-clockStart) /
1000000.0) + " millisecs");
    }
    connector.close();
}
```

Listing 0.5 Module removeStopWordsAndStem

```
public static String removeStopWordsAndStem(String input) throws
IOException {
    TokenStream tokenStream = new StandardTokenizer(
            Version.LUCENE_30, new StringReader(input));
    tokenStream = new StopFilter(true, tokenStream, stopwords);
    tokenStream = new PorterStemFilter(tokenStream);
    StringBuilder sb = new StringBuilder();
    TermAttribute termAttr = tokenStream.getAttribute(TermAttribute.class);
    while (tokenStream.incrementToken()) {
        if (sb.length() > 0) {
            sb.append(" ");
        }
        sb.append(termAttr.term());
    }
    return sb.toString();
}
```

Listing 0.6 Module parseDocument

```
public void parseDocument(File doc) throws ParserConfigurationException,
SAXException, IOException {
    //get a factory
    SAXParserFactory spf = SAXParserFactory.newInstance();
    //get a new instance of parser
    SAXParser sp = spf.newSAXParser();
    //parse the file and also register this class for call backs
    sp.parse(doc, this);
    connector.close();
}
```

Listing 0.7 Module splitPhraseBigram

```
public static List<ByteBuffer> splitPhraseBigram(String searchString,String
fileName) throws FileNotFoundException, IOException {
    List<ByteBuffer> results = new ArrayList<>();
    stopwords = readStopWordSet(fileName);
    String[] s = searchString.toLowerCase().split(sDelimiter);
    s = removeStopWordsAndStem(s);
    for (int i = 0; i < s.length - 1; i++) {
        results.add(ByteBuffer.wrap((s[i] + ":" + s[i + 1]).getBytes()));
    }
    return results;
}
```

Listing 0.8 Module splitPhraseTrigram

```
public static List<ByteBuffer> splitPhraseTrigram(String
searchString,String fileName) throws FileNotFoundException, IOException {
    List<ByteBuffer> results = new ArrayList<>();
   stopwords = readStopWordSet(fileName);
    String[] s = searchString.toLowerCase().split(sDelimiter);
    s = removeStopWordsAndStem(s);
    if (s.length >= 3) {
        for (int i = 0; i < s.length - 2; i++) {
            results.add(ByteBuffer.wrap((s[i] + ":" + s[i + 1] + ":" + s[i
+ 2]).getBytes()));
        }
    }
    return results;
}
```

Listing 0.9 Module splitPhraseUnigram

```
public static List<ByteBuffer> splitPhraseUnigram(String
searchString,String fileName) throws FileNotFoundException, IOException {
    List<ByteBuffer> results = new ArrayList<>();
    stopwords = readStopWordSet(fileName);
    String[] s = searchString.toLowerCase().split(sDelimiter);
    s = removeStopWordsAndStem(s);
    for (int i = 0; i < s.length; i++) {
        results.add(ByteBuffer.wrap(s[i].getBytes()));
    }
    return results;}
```

References

[HEWITT, 2011]

Eben HEWITT, Cassandra: The Definitive Guide, O'Reilly Media, Inc., 2011

[KIM, 2008]

Min-Soo KIM, Kyu-Young WHANG, Jae-Gil LEE, Min-Jae LEE, Structural optimization of a full-text n-gram index using relational normalization, *The VLDB Journal* (2008) 17:1485–1507

[Lucid, 2009]

Lucid Imagination, What Lucene and Solr Open Source Search can do for Enterprise Search, April 2009

[MAHMOUD, 2011]

Rammal MAHMOUD, Sanan MAJED, Improving Arabic Information Retrieval System using n-gram method, WSEAS Transactions on Computers, Volume 10 Issue 4, April 2011, Pages 125-133

[SADALAGE, 2013]

Pramod J. SADALAGE, Martin FOWLER, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2013

[WILSON, 2008]

Theresa WILSON, Stephan RAAIJMAKERS, Comparing word, character, and phoneme n-grams for subjective utterance recognition, ISCA, 2008

[YAMAMOTO, 2003]

Hiroshi YAMAMOTO, Seishiro OHMI, Hiroshi TSUJI, Incremental Indexing and Its Evaluation for Full Text Search, Idea Group Inc., 2003

[ZICARI, 2012]

Roberto V. ZICARI, Integrating Enterprise Search with Analytics. Interview with Jonathan Ellis, 16.04.2012

Web Resources

[1] Adam Pullen, Final concept, http://www.finalconcept.com.au/article/view/apache-solr-hints-and-tips (last visited on 10.10.2012)

[2] Cassandra Wiki, GettingStarted, http://wiki.apache.org/cassandra/GettingStarted (last visited on 13.05.2013)

[3] chco (alias), Replica Placement Strategies When Using Cassandra, O'REILLY Answers, 22 Jan 2011, http://answers.oreilly.com/topic/2408-replica-placement-strategies-when-using-cassandra/ (last visited on 13.05.2013)

[4] Chrisumbel, Solr/Lucene for SQL server, http://www.chrisumbel.com/article/lucene_solr_sql_server (last visited on 10.10.2012)

[5] Community Help Wiki, Packaging webapps for deployment in Tomcat 6.0 in Debian and Ubuntu, https://help.ubuntu.com/community/Tomcat/PackagingWebapps (last visited on 15.10.2012)

[6] Datastax, About Data Partitioning in Cassandra, Apache Cassandra 1.0 Documentation, http://www.datastax.com/docs/1.0/cluster_architecture/partitioning (last visited on 14.05.2013)

[7] Datastax, Initializing a multiple node cluster, Apache Cassandra 1.2 Documentation, http://www.datastax.com/docs/1.2/initialize/cluster_init#cluster-init-multiple (last visited on 13.05.2013)

[8] Diversions, Introduction to Cassandra, 5th October 2012, http://dafreels.wordpress.com/ (last visited on 14.05.2013)

[9] Edlich S., NoSQL, http://nosql-database.org/ (last visited on 08.08.2013)

[10] Featherston D., Cassandra: Principles and Application, in the author's graduate work at the University of Illinois at Urbana-Champaign, 03 Aug 2010, http://d2fn.com/2010/08/03/cassandra-paper.html (last visited 18.05.2013)

[11] Feipeng L. (alias roman10), Apache Cassandra Understand Replication, August 24, 2012, http://www.roman10.net/apache-cassandra-understand-replication/ (last visited 18.05.2013)

[12] Gaucherin, B., solr configuration, http://bengaucherin.wordpress.com/2011/09/17/one-more-solr-step-by-step-setup-tutorial-%E2%80%93-part-ii-of-ii/ (last visited on 03.10.2012)

[13] Hulen, Cassandra Performance Testing on EC2, Sept 2010, http://www.hulen.com/post/22803493165/cassandra-performance-testing-on-ec2 (last visited on 03.07.2013)

[14] Lucidimagination, What Is Indexing?, http://lucidworks.lucidimagination.com/pages/viewpage.action?pageId=9241300 (last visited on 10.10.2012)

[15] McGee M., Facebook: 3.2 Billion Likes & Comments Every Day, Marketing Land, 27 Aug. 2012, http://marketingland.com/facebook-3-2-billion-likes-comments-every-day-19978, (last visited on 10.07.2013)

[16] Putnam C., Faster, Simpler Photo Uploads, Le blog Facebook, 5 février 2010, http://www.facebook.com/blog/blog.php?post=206178097130, (last visited on 10.07.2013)

[17] Querna P., Cassandra Token Selection, http://journal.paul.querna.org/articles/2010/09/24/cassandra-token-selection/ (last visited on 01.05.2013)

[18] Reagan D., Installing Cassandra on Ubuntu Linux, http://dustyreagan.com/installing-cassandra-on-ubuntu-linux/ (last visited on 09.10.2012)

[19] Stack Overflow, Cassandra - Understanding Rack Concept on PropertyFileSnitch example, http://stackoverflow.com/questions/13882313/cassandra-understanding-rack-concept-on-propertyfilesnitch-example (last visited on 13.05.2013)

[20] The Apache Software Foundation, Apache Solr, http://lucene.apache.org/solr/ (last visited on 08.08.2013)

[21] The Apache Software Foundation, Tomcat Web Application Deployment, http://tomcat.apache.org/tomcat-6.0-doc/deployer-howto.html (last visited on 15.10.2012)