W205 - Project

Enterprise search engine for log data

Vineet Gangwar and Anant Srivastava

Introduction

As IT infrastructure becomes increasingly large and complex, troubleshooting problems or analyzing user behaviour means combing through massive amounts of log data from multiple systems. Traditional methods of 'cat' and 'grep' are becoming inadequate.
The ability to process log data is one of the central use cases of big data. As log data is inherently unstructured text, so Information Retrieval techniques are best suited for processing log data. Log data is also being increasingly recognized as a gold mine that provides end-to-end visibility across the IT estate.

In this project we built an enterprise search engine for log data that uses Inverted Index for indexing and Boolean search for query processing. Our search engine for log data can also be considered as a specialized database. From a datastore/database point of view, the search engine is geared towards delivering performance in reads and inserts. Updates will not be supported as log data does not need to be updated

Our objective is to provide a handy tool to the system administrator whose job is to troubleshoot complex systems by analyzing logs. With this objective in mind we developed two versions of our search engine.

1. A single system implementation
   The objective of the single system implementation is to provide a single-pane-of-glass (interface) and high speed access to log data for day to day troubleshooting needs. This type of troubleshooting requires access to relatively current log data (i.e. logs not older than 7 days) but the sources of logs are very high (i.e. logs generated by many applications running on tens or hundreds of systems). This implementation is much faster than the traditional method of 'cat' and 'grep'
2. A hybrid system that runs partly on AWS (EMR and S3) and partly on a local system
   The objective of this version is to provide a single-pane-of-glass access to extremely high volume of logs. Extreme high volumes is possible because the indexing process uses AWS EMR for index processing and S3 for log storage

There are many log management products available. The top names are Splunk, Sumo Logic, Logstash and Loggly. Log management as a field is very versatile. It is used as a platform for many use cases such as security management, IT Operations and fraud detection. Data science applications (such as machine learning) on top of log management products is an exciting new frontier that will see years of innovation.

The data

Log data has high volume and high velocity. Large organizations with around 10K server instances can generate upwards of 1 TB of data a day with bursts reaching up to several hundred MBs per second. Log data can have variety also such as XML, free text and semi-structured text. Log entries generally have the following information – TimeStamp, DeviceName, ApplicationName, Severity and MessageText. For our project we have considered log data to be a string of text (i.e. words separated by spaces) terminated with a new line. Each log entry has 10 words.
Due to the nature of the input file we used, we left out the following trivial tasks:

1. Punctuation removal
2. Stop words removal
3. Changing case

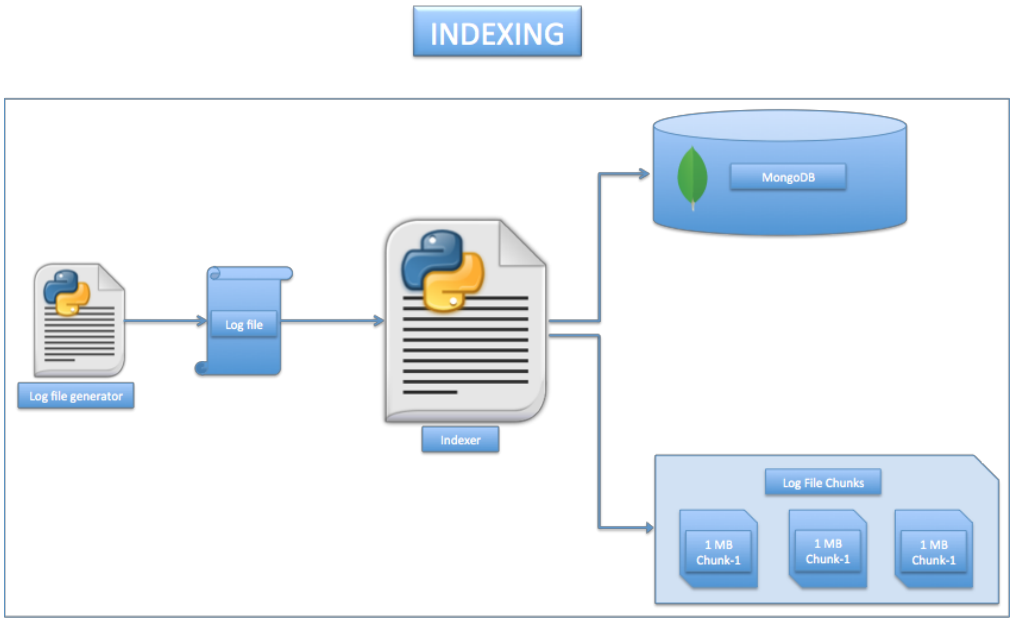
Solution Architecture

Single system implementation

In the single system implementation, all the components reside in a single system. The solution consists of two parts:

1. Indexing module
2. Boolean Search module

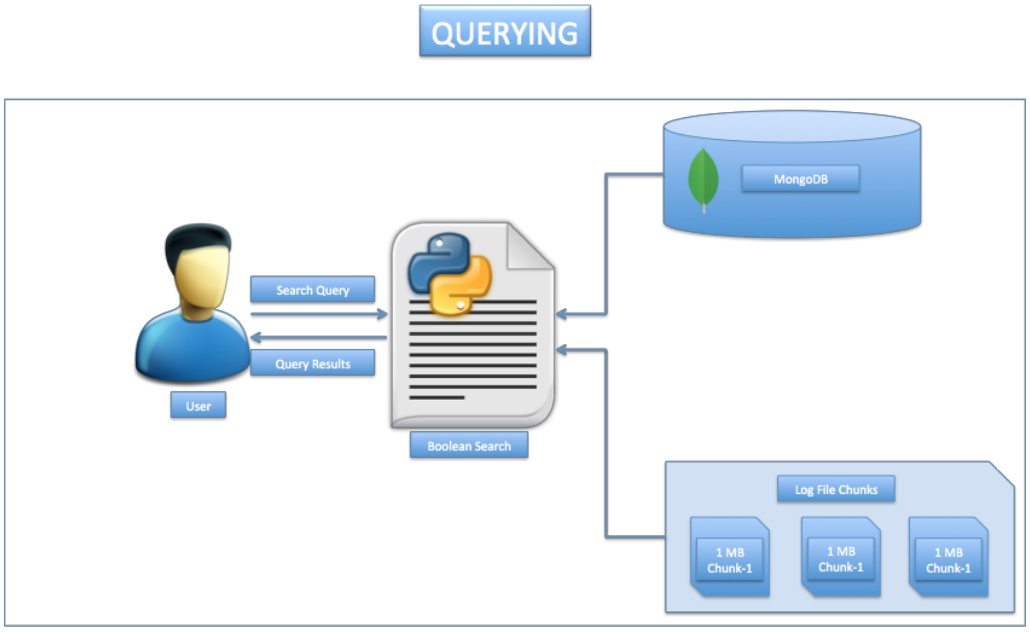
Indexing Module Architecture



Component	Description
Python script to generate log file	Our initial plan was to write a script to generate the input log file. In one of the class activities our professor gave us a script that generates flat text files from a corpus of ~25K words. Since writing this script is not central to our project, so we decided to use the script given by our professor.
Log file	The log file that we used in our project was generated by the script mentioned above. It consisted of 1 million lines and each line consisted of 10 words separated by space. The size of the input log file is ~83 MB
Python script to create index	This script is the core of the indexing module. It creates the term dictionary, the postings list and breaks up the input log file into 1 MB chunks. It uses bulk insert method of MongoDB to achieve faster inserts.
MongoDB	The term dictionary and the postings list is stored in MongoDB. The postings list consists of the document name, the offset of the log entry in the 1MB chunk file and the length of the log entry. e.g. postings list - `{'term':`

	`word1, {'documentName': docName, 'logOffset': logOffset, 'logLength': logLength}}`
1 MB log file chunks	The size of the chunk files is configurable. Small chunk files enables faster access of individual log entries

Boolean Search Module Architecture



Component	Description
User	The user submits the boolean search in a query language that handles any combination of the logical operators - AND, OR and NOT. Also based on available memory, it can handle any number of terms
Query	e.g. search query: `(((term1 & term2) | term3) ! Term4)` where & = AND; | = OR ! = NOT
Python script for Boolean Search	This python script is the core of the Boolean search module. It processes the query and returns the result to the user. It supports 2 query modes - it returns either just the count of the result or the count along with individual log entries. This script utilizes python set methods of Intersection, Union and Difference for boolean AND, OR and NOT
Shunting Yard Algorithm	The boolean search script uses the shunting yard algorithm to process the query by converting it from infix to postfix notation
MongoDB	Contains the terms and the postings list.

1 MB log file chunks	The boolean search script uses the postings list to read the individual log files from the chunk files

Hybrid System Architecture

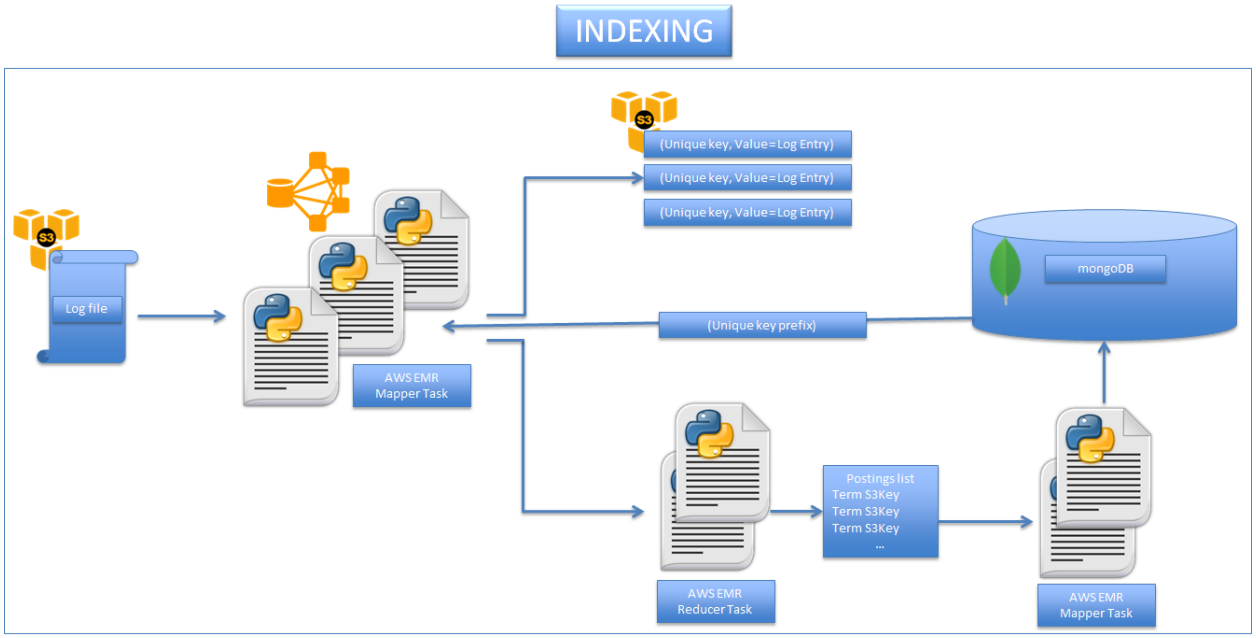
The solution consists of 2 parts:

1. Indexing module comprising of index generation and storage.
2. Boolean Search module

The objective of the hybrid system is to provide scalability on two fronts:

1. Parallel processing of input log files to generate Index
2. Ability to store large volumes of logs as the logs are stored on AWS S3

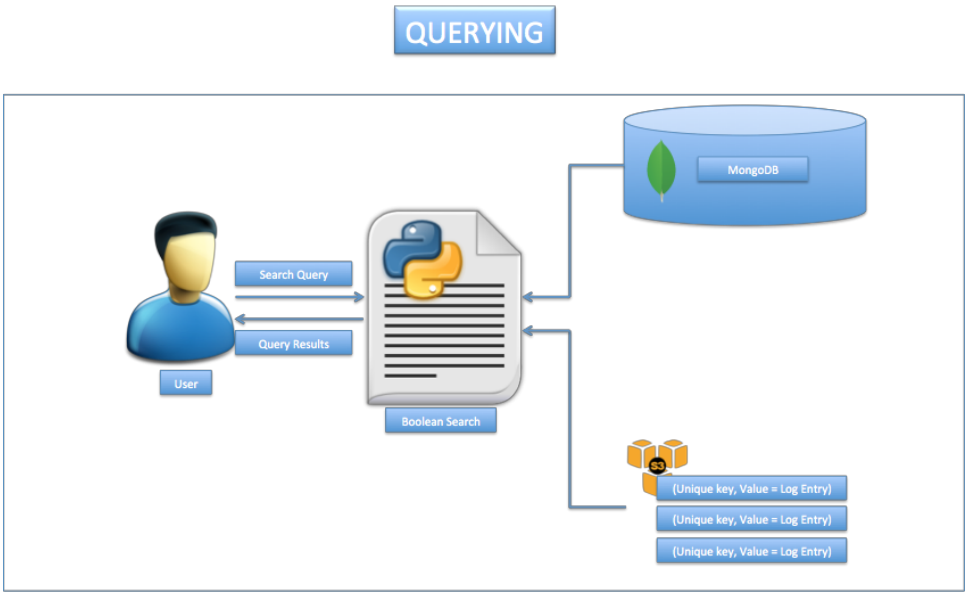
Indexing Module Architecture



Component	Description
S3 - Input Log File	The indexing process requires the input log files to be available in AWS S3. So user has to first load all the log files into AWS S3
AMS EMR - 1st MapReduce task	The indexing process uses AWS EMR to parallelize part of index creation. It uses a map reduce job. The AWS EMR framework reads the input log file from S3 and feeds the log entries line by line to the mapper task. The mapper task generates a unique key for every log entry and uses the unique key to store each log entry individually in S3 as a unique key value pair. It then splits each log entry into words and outputs each word along with the corresponding S3 key. The reducer aggregates the output and writes to S3 for insert into MongoDB.

S3 - Log Storage	The indexing process uses S3 for log storage. This gives the ability to store massive amounts of data. E.g. Key: `1a1234` It comprises of two parts: • A prefix retrieved from database. • A sequence number of the log processed by the mapper task
S3 - Postings List	Each mapper task generates a postings list that is also stored in S3 in the format: `Term, S3KeyName` list. Example postings list: `[{"Term": "One"}, {"$addToSet": {"postingList": {"$each": ["49a5770","49"]}}}]`
2nd Mapper Task - Postings list from S3 to MongoDB	This mapper task reads the postings list from S3 and then upserts into MongoDB
MongoDB	MongoDB is used to store the postings that the search module uses to respond to user queries. It also provides the unique prefix to the mapper tasks. The unique prefix is used by the mapper task to create unique S3 key

Boolean Search Architecture



Component	Description
User	The user submits the boolean search in a query language that handles only OR operators for now.
Query	e.g. search query: `term1 term2 term3`

Python script for Boolean Search	2 python scripts comprise the search module. One of the scripts returns the counts and the other returns the logs
MongoDB	Contains the terms and the postings list.
S3 - Log Storage	The boolean search script uses the postings list to read the individual log files from S3

Implementation

Single System - Indexer

Technologies/features/concepts

Technology/Features/ Concepts	Description
Python	Python is the language of choice as it one of the primary languages used in the MIDS program. It uses pymongo module for interaction with MongoDB
Inverted Index	Inverted Index is the main concept on which this project was conceived of and developed
Dictionary data structure	Dictionary data structure is used for storage and processing of postings list
Set data structure	The Set data structure is used for speed improvements. In Python conversion from list to sets removes duplicates. This deduplication, instead of using integrity constraints in MongoDB helps speed up processing
File IO Seek method	This is another key feature that is used in this project. The postings list stores the Document name, offset and log length of each log entry in the 1 MB chunk files. The seek method is used to position the file pointer directly at the start of the log file
MongoDB	MongoDB is used to store the Postings list. The solution is implemented using two separate Python scripts - Indexer and Searcher. To avoid the additional complexity of inter-process communication, we have used MongoDB. The indexer writes to MongoDB and the searcher reads from MongoDB
MongoDB Bulk Insert	We are using Bulk inserts of 10K postings list to achieve faster inserts into MongoDB. This is a configurable parameter. Bulk insert is done through Unordered Bulk operation of MongoDB

Postings List

Structure

Postings List of a term is a list of dictionaries. Each posting refers to a single log entry in the 1MB Chunk files. Each posting consist of the following:

Document Name	Name of document that contains log entries that contains the term
Offset	The start position of the log entry in the chunk file

Log length	The length of the log entry

Storage

Postings lists are stored in a MongoDB collection. Each document in the collection contains only 2 key value pairs

Key	Value
Term	A single term
Postings List	A single dictionary instance of a single postings list

Overcoming 16 MB MongoDB document size limit

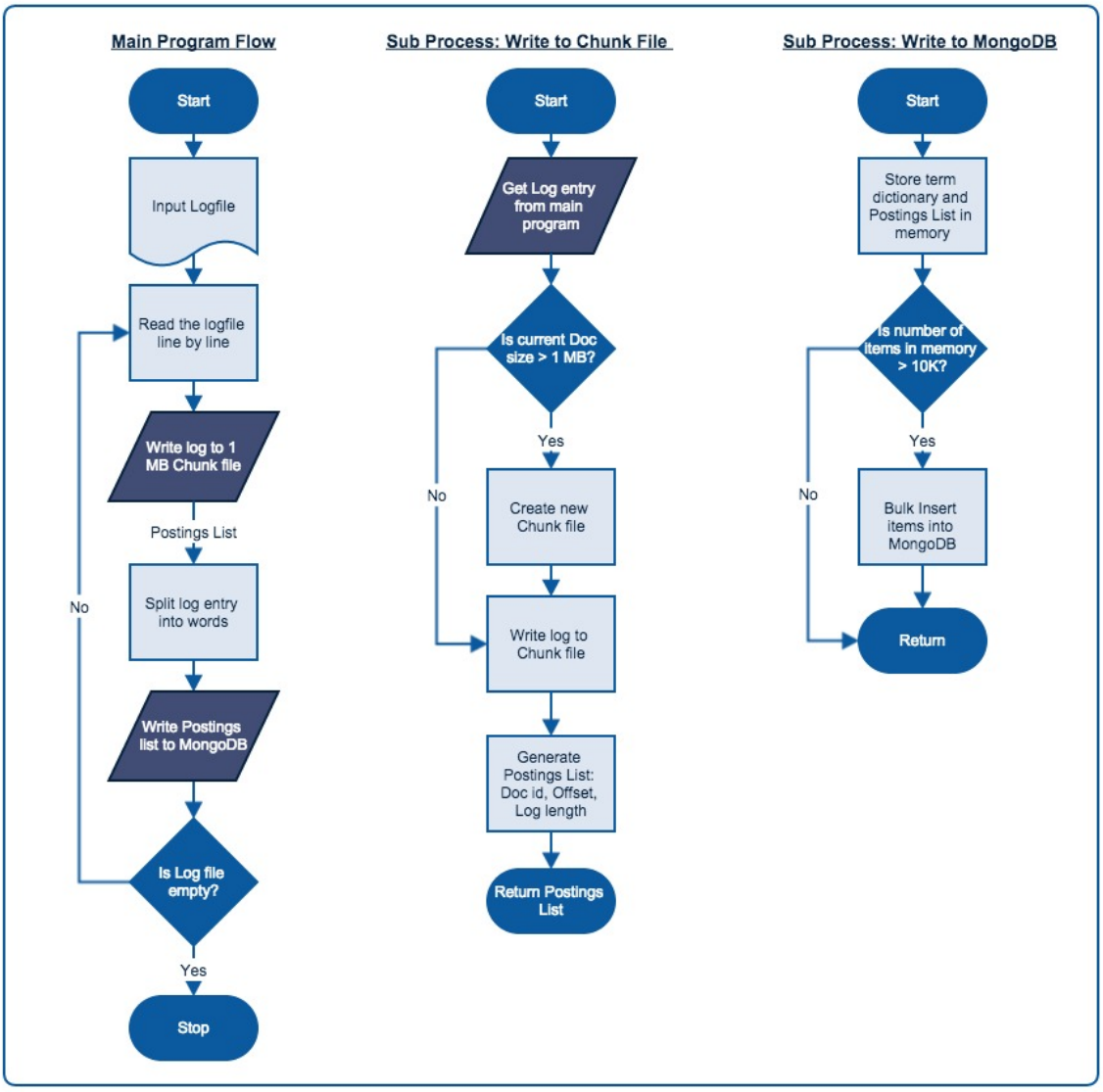
If a term exists in two log entries then for that term there will be two documents in the MongoDB collection. The postings list of each document points to the two log entries in the 1 MB chunk files. This allows us to overcome the 16MB MongoDB document size limit.

MongoDB indexing

The MongoDB collection is indexed on the key ‘Term’. This allows for faster reads

FlowChart

The flowchart of the Indexer is given below. 2 classes are used to implement this module



Single System - Boolean Search

Technologies/features/concepts

Technology/Features/ Concepts	Description
Python	Python is the language of choice as it one of the primary languages used in the MIDS program. It uses pymongo module for interaction with MongoDB
Shunting Yard Algorithm	One of the key features of our solution is a query language using which users can submit queries with any combination of the boolean operators AND, OR and NOT. The shunting yard algorithm makes this possible. We

	use this algorithm to convert the query from infix notation into postfix notation.
Processing Postfix	The stack data structure is used to process the postfix notation to generate results for the user
Set Methods	Boolean AND, OR and NOT and implemented using the Intersection, Union and Difference methods respectively of the the Python set object
MongoDB	MongoDB is used to store the Postings list. The solution is implemented using two separate Python scripts - Indexer and Searcher. To avoid the additional complexity of inter-process communication, we have used MongoDB. The indexer writes to MongoDB and the Boolean Search module reads MongoDB
File IO Seek method	This is another key feature that is used in this project. The postings list stores the Document name, offset and log length of each log entry in the 1 MB chunk files. The seek method is used to position the file pointer directly at the start of the log file and the file read method is used to exactly a single log file using the log length from the postings list
Stack data structure	The stack data structure is central to query processing. It is used both during infix to postfix conversion and during postfix processing.

The Query language and Query Processing

One of the key features of our solution is a query language using which users can submit queries with any combination of the boolean operators AND, OR and NOT. Example query:

```
( ( ( ( ( term1 & term2 ) | term3 ) ! term4 ) & term5 ) ! ( term6 | term7 ) )
```

where & = AND; | = OR; ! = NOT

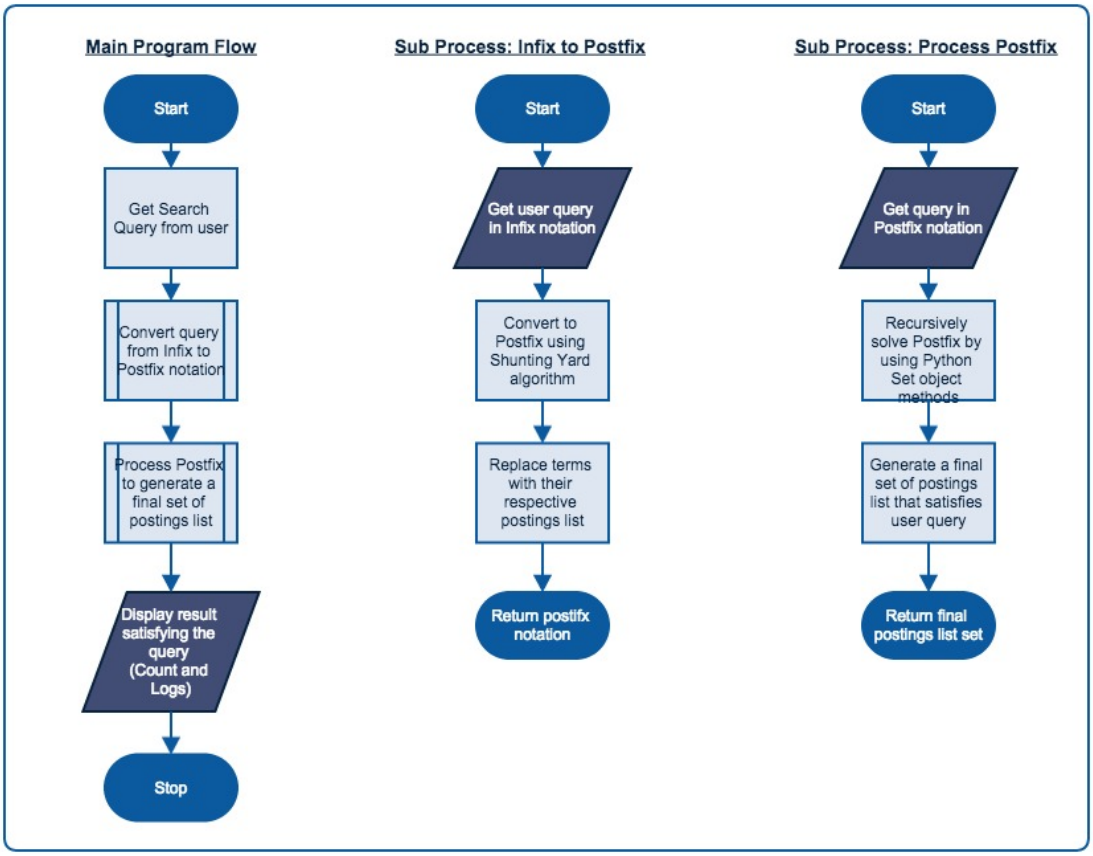
Query processing is accomplished in 2 steps:

1. The first step converts the query from infix notation to postfix notation. In the postfix notation, each term is replaced by its set of all postings list. This postings list set is obtained by querying MongoDB
2. In the second step, the postfix notation is solved using python set object's methods of Intersection, Union and Difference. The output of this step is a single set of postings list that satisfies the query submitted by the user.

The final set of postings list obtained above is used to return the count and the log entries to the user

Flowchart

The flowchart of the Indexer is given below. 4 classes are used to implement this module.



Single System - Code

The following explains the code:

Python Script Name	Description
Single_System_Indexer.py	- This script splits the input log files into 1 MB chunk files - Splits each log/line to create term dictionary - Generates postings list for each term - Inserts the postings list into MongoDB - The code contains appropriate comments
Single_System_BooleanSearch.py	- This script receives query from the user - Converts the query from infix to postfix notation using shunting yard algorithm - Replaces each term in the postfix notation with its set of postings list - Uses python set object methods of Intersection, Union and Difference to process the postfix notation to generate the final set of postings list - Counts the items in the final postings list set to return count - Uses the final postings list set to query 1 MB chunk files to return logs that satisfy the user's query

	- User query format has some rough edges. The script expects the query in the following format: ((term1 | term2) & term3) Every element should be separated by space Every pair of logical terms should be in parenthesis - The code contains appropriate comments

Hybrid System - Indexer

Technologies/features/concepts

Technology/Features/ Concepts	Description
Python	Python is the language of choice as it one of the primary languages used in the MIDS program. It uses pymongo module for interaction with MongoDB, Boto to connect to S3 and random module to generate random numbers
AWS EMR	The task of processing logs to generate postings list, very neatly fits into the map-reduce paradigm. That is why we chose to AWS EMR because it is very easy to use EMR to run map-reduce jobs. EMR also allows us to perform parallel upserts of posting list into Mongodb.
S3	We chose S3 as our log store because S3 provides the following advantages: 1. Massive amounts of logs can be stored in S3 2. S3 actually makes the implementation simpler. The postings list just needs to contain the S3Key rather than the document name, offset and log length; unlike the single system implementation
Inverted Index	Inverted Index is the main concept on which this project was conceived of and developed
MongoDB	MongoDB is used for 2 things: 1. To store the Postings list 2. To generate unique id prefix for S3 keys
MongoDB Set functions	These functions allow us to do incremental additions to the posting list and perform intersection/union among the posting lists

Unique Key generation

The solution generates a unique key for every log entry that is stored in S3. The unique S3 key is made up of two parts:

1. Part 1: Is a prefix that is provided by MongoDB to each mapper task
2. Part 2: A sequence number of each log processed by each mapper

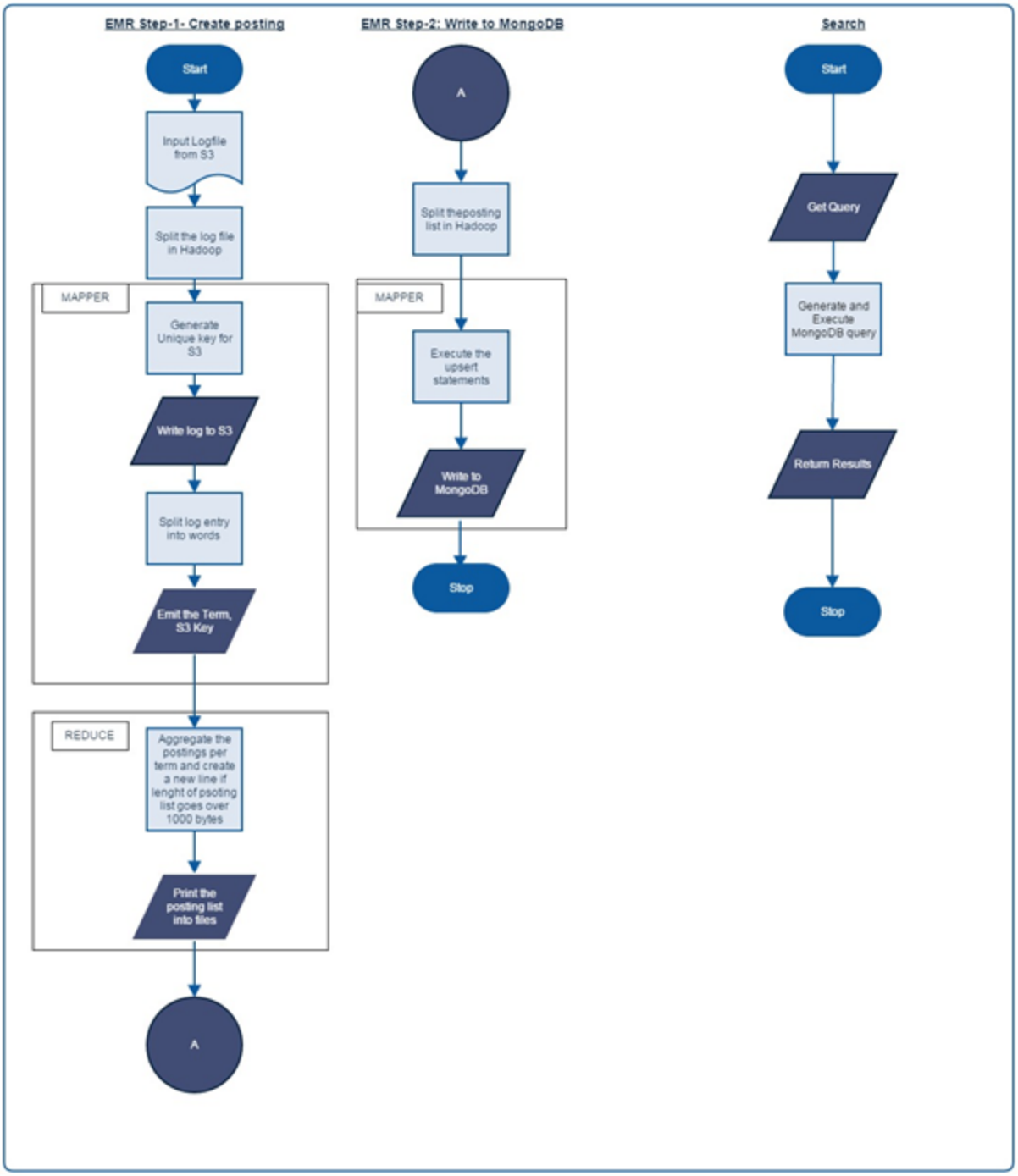
At startup each mapper queries a single document collection in MongoDB. That document contains the prefix. After reading the prefix, the mapper task increments the prefix. All mapper tasks perform the same operations and hence each get a unique prefix. The operation in MongoDB is atomic ie each mapper has

exclusive access to the document at a time. Since each mapper accesses the same document, so we each mapper gets a unique prefix to use to generate a unique key for each log across all mappers.

Example S3Key: `1a234`

Flowchart

The flowchart of the Indexer and Search is given below:



Hybrid System - Boolean Search

The hybrid System has 2 modules that support for boolean 'OR' operations. The search module returns the matching logs and count module returns the count of matching records. Note that at this point the hybrid search module only implements OR operation. The search terms are supplied from the command line and parameterized into the search query for mongodb. OR operation is achieved using the aggregation pipeline of MongoDB.

The rest of this module is simpler than the single system module. It is simpler because the postings list is simpler. The single system postings list contains 3 components - Document Name, Offset and Log length. In comparison the hybrid system's postings list has a single component - just the list containing S3Keys.
Once the search module retrieves the S3Keys from MongoDB, it queries S3 using Boto to retrieve the log entries.

Hybrid System - Code

The following explains the code:

Python Script Name	Description
Hybrid_System_IndexMapper.py	- This is the Mapper class. It connects to Mongodb to fetch the prefix and then iterates over each line supplied by AWS EMR. Subsequently it creates a S3 key and stores the log entry for that S3 key. It also uses nltk corpus to omit stop words when emitting the words and S3 key for that line.
Hybrid_System_IndexReducer.py	- The reducer receives terms along with the S3 keys and aggregates all the keys for a term into a posting list and prints the term and its posting list to a file.
Hybrid_System_MongoUpdateMapper.py	- This Mapper takes files from the previous reduce step as input and for each line performs upserts into Mongodb
Hybrid_System_SearchCount.py	- This search script does not need to access s3 key since it only returns the counts from mongodb directly.
Hybrid_System_Search.py	- This script returns the logs from s3. It first executes the search in mongo db and fetches the posting list and subsequently gets the corresponding values from s3.

Results

We compared the performance of both the single system and the hybrid system against grep. We conducted 5 sets of tests on an input data that than 1 million lines with 10 words per line:

1. The first test was using a single term
2. The second test was an OR of 2 terms
3. The third was an OR of 3 terms
4. Similarly there was a fourth test
5. And a fifth test

For each of the tests above, the following was measured:

1. Time taken by the single system solution to return count
2. Time taken by the grep to return count
3. Time taken by the single system solution to return logs
4. Time taken by the grep to return logs
5. Time taken by the hybrid system solution to return logs

Here are the commands that was executed:
For Grep:

```
time cat Input_Data/random-0.txt | grep -w -c "chore"
time cat Input_Data/random-0.txt | grep -w -c "chore\|bland"
time cat Input_Data/random-0.txt | grep -w -c "chore\|bland\|out"
time cat Input_Data/random-0.txt | grep -w -c "chore\|bland\|out\|till"
time cat Input_Data/random-0.txt | grep -w -c "chore\|bland\|out\|till\|shown"
```
To return logs, the -c option from grep was removed

For Single System solution:
```
time python single_search.py "( chore )"
time python single_search.py "( chore | bland )"
time python single_search.py "( ( chore | bland ) | out ) "
time python single_search.py "( ( ( chore | bland ) | out ) | till )"
time python single_search.py "( ( ( ( chore | bland ) | out ) | till ) | shown
)"
```

For Hybrid System solution
```
time python hybrid_searchS3.py "( chore )"
time python hybrid_searchS3.py "( chore | bland )"
time python hybrid_searchS3.py "( ( chore | bland ) | out ) "
time python hybrid_searchS3.py "( ( ( chore | bland ) | out ) | till )"
time python hybrid_searchS3.py "( ( ( ( chore | bland ) | out ) | till ) |
shown )"
```

Both the single system and hybrid system search module was run twice. Once to return count and once to return the logs. This was done by commenting out the code in the script that returns the logs.

Single System Implementation

On average the single system is considerably faster than grep. On average it returned counts 52 time faster than grep; and returned logs 34 times faster than grep.

The tests were done on Ubuntu running on a Virtual Machine running on MacBook Pro.

System details

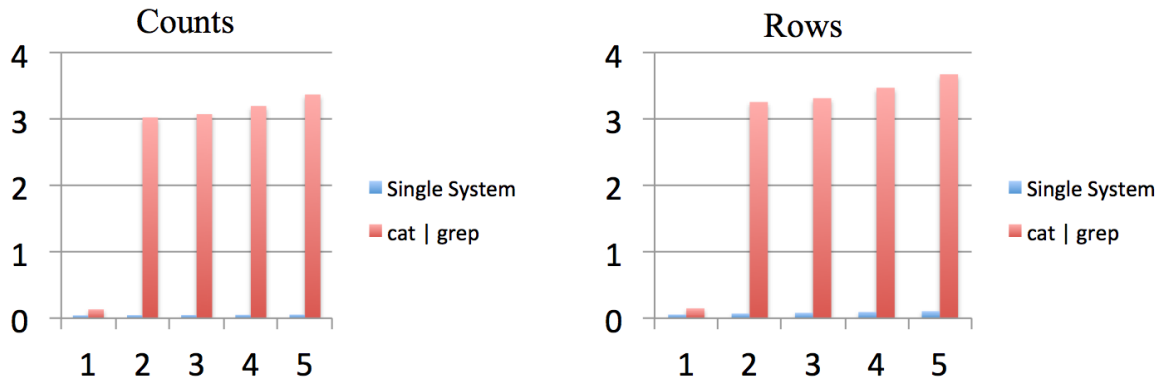
MacBook Pro	
Processor	2.2 GHz Intel Core i7
Cores	8
Memory	16 GB 1600 MHz DDR3
OS	OS X Yosemite 10.10.3
Flash Storage	256 GB
Internet	15 Mbps

Ubuntu	
Version	14.04 LTS
Cores	4
Memory	4 GB
VMWare Fusion	7.1.1
Python	2.7.6
MongoDB	3.0.2

Measurements in secs

# of terms	Rows returned	Counts		Logs	
		Single Sys	Grep	Single Sys	Grep
1	~350	0.044	0.134	0.056	0.149
2	~750	0.047	3.029	0.071	3.261
3	~1150	0.049	3.078	0.082	3.319
4	~1550	0.052	3.201	0.094	3.476
5	~1900	0.055	3.374	0.106	3.679
	Average:	0.0494	2.5632	0.0818	2.7768

This shows that the single system was faster than grep by ~52 times for returning counts and ~34 for returning logs.



Note:
X Axis = Number of terms with OR operator; Y Axis = Time in seconds
Left Panel = Only counts returned; Right Panel = Rows returned

Additional measurements

Single System	
Time taken to Index and load into MongoDB	3 mins 54 sec

Hybrid System Implementation

The hybrid system was much faster than grep when returning counts but it was considerably slower than grep when returning the actual logs and a lot of that latency can be attributed to the I/O involved between AWS and local system.

The tests were done on MacBook Pro with the Macbook connecting to AWS for retrieving logs.

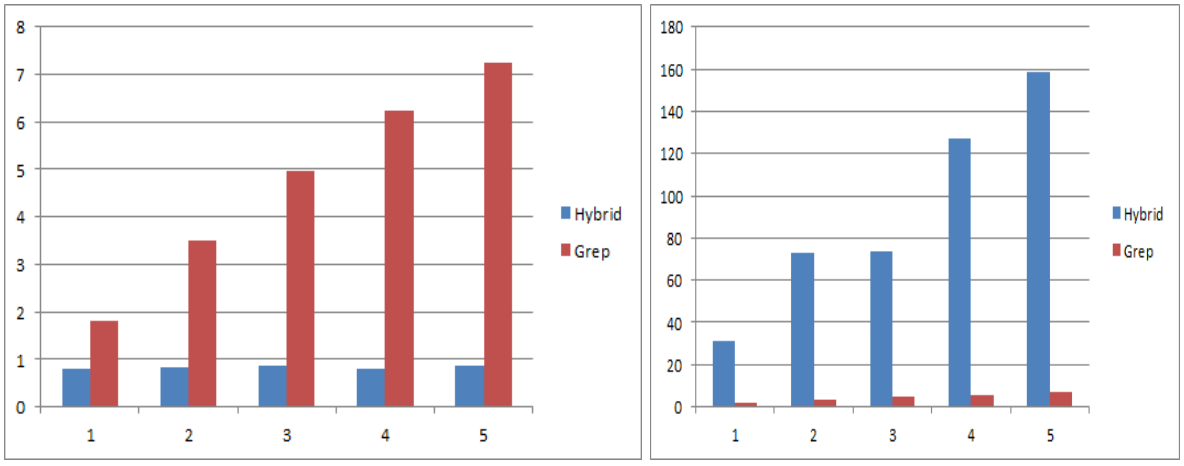
System details

MacBook Pro	
Processor	2.6 GHz Intel Core i5
Cores	4
Memory	8 GB 1600 MHz DDR3
OS	OS X Yosemite 10.10.3
Flash Storage	128 GB
Internet	15 Mbps

Measurements in secs

# of terms	Rows returned	Counts		Logs	
		Hybrid Sys	Grep	Hybrid Sys	Grep
1	~360	.80	1.82	31.23	1.81
2	~800	.82	3.51	72.7	3.27
3	~1150	.88	4.96	73.7	4.82
4	~1550	.81	6.25	127.2	5.91
5	~1900	.88	7.23	158.6	6.97
	Average:	.838	4.75	92.6	4.55

This shows that the hybrid system for 5.6 times faster than grep for counts but 20 times slower for logs.



Note:
X Axis = Number of terms with OR operator; Y Axis = Time in seconds
Left Panel = Only counts returned; Right Panel = Rows returned

Additional measurements

Hybrid System	
Time taken to Index (1 large Master and 4 large cores)	1 hr 17 minutes
Mongodb upserts (1 large Master and 4 large cores)	8 minutes

Conclusion

Our project is a success against our original objectives. We had two objectives:

1. Use the concept of Inverted Index to provide faster access to logs than grep
2. Use available cloud technologies for scalability - Parallel processing of input logs for indexing and storage of massive amounts of logs

We were able to achieve both the objectives. On average our solution is 52 times faster than grep for returning counts and 34 times faster for returning logs. Our solution also transparently logs into AWS S3 and retrieves logs based on user query

We conclude that using a simple concept like inverted index, it is possible to build fast databases. Additionally, the available cloud technologies such as AWS EMR and S3 are mature enough for any organization to build a home grown super scalable database/search engine without having the need to invest or manage complex products

We had also initially planned to measure the performance of our solution against the commercial product Splunk. We were unable to do so because Splunk does not have a version for Ubuntu. We have done all of our performance tests on Ubuntu.

The Hybrid system was slower than grep for our test input log size of ~83 MB. However, we believe that for larger log files and for queries that return smaller number of logs/rows our hybrid system will be faster than `cat | grep`.

Challenges

Unique Key generation for S3

All logs are are individually stored in S3. This requires generating a unique key for all the logs. Generating a key that is unique across all the Mapper tasks was a challenge because the mappers run independently.

To overcome this problem, we maintained our prefix sequencing in Mongodb and each mapper task used the prefix sequence obtained from the database for all keys generated by that task.

Document size in MongoDB

Single System

Our initial plan was to store each term and its complete postings list in a single document ie. we wanted to mirror the inverted index data structure in MongoDB. However, the 16 MB document size limit of MongoDB prevented us from implementing it. The solution was to store each posting list in a different row.

Hybrid System

The 16 MB limit for the size of a posting list for a term still applies to the hybrid system.

Improvements

1. Considering the way we are storing the index - one row for a term and a postings list - maybe a columnar database might result in improving the performance
2. Another area of improvement is to consider compression techniques for the index
3. By just storing the location of a term inside a log in the postings list we can extend the system to support multi-word searches such as (`"term1 term2 term3" & term4`)
4. AWS
    a. EBS vs S3: EBS is supposedly much quicker than S3 and using EBS would most likely speed up our search queries.
    b. Search performance on EC2: The python search script is currently executed on the local system. We would like to execute the search script on the EC2 instance and output the search results to a file and see whether the system performs better. We are expecting the performance to be between because of low network latency between EC2 and S3
5. MongoDB
    a. mmap v1 vs wiredtiger: mmap v1 is the default mongodb kernel that we used and it does not allow for concurrent access to the collection. Wiredtiger is another kernel that allows for concurrent access and compression since Wiredtiger has more control over the memory. We can evaluate how the overall process works with wiredtiger.
    b. GridFS: GridFS is a new introduction to mongodb and allows for documents to be larger than 16 MB. There is not enough information available at this time about this new feature but we would like to evaluate this feature to overcome the 16 MB posting list problem.
    c. Indexing and Sharding: Sharding and Indexing on the posting list would benefit certain type of queries. We would like to explore this feature further and see how it would impact the searches.

Limitations

Single System

We didn't implement the following trivial though important tasks of text processing because the nature of the input file that we used:

1. Punctuation removal
2. Stop words removal
3. Changing case

The query language some rough edges. In the form that we have implemented, the following has to be kept in mind:

1. Every element in the query (i.e. either parenthesis, term or boolean operator) has to be separated by space
2. Every two terms have to in parenthesis as shown above:

e.g. `term1 & term2 & term3` has to be written as
`( ( term1 & term2 ) & term3 )`

Of course with a few lines of additional coding the query language can be made more intuitive for the user

Hybrid System

1. Hybrid system takes care of stop words and works in the UTF-8 mode. It does not take care of punctuations though.
2. The query tool for the hybrid implementation needs more work to support and,or, not type of queries.