Universität Rostock Traditio et Innovatio

2018

Data Manipulation at Scale: Systems and Algorithms

By: Mohammad Ghojal
Tutor: Meisam Booshehri
Professor: Peter Luksch

Abstract:

In this project, it is showing the landscape of parallel databases, distributed systems, and programming languages. The principle on which they rely, their tradeoffs, and how to evaluate their utility against the needs. Also it describes practical systems were derived from the frontier of research in computer science and what systems are coming on the horizon. Cloud computing, *SQL*, and *NoSQL* databases, *MapReduce*, *Spark* and its contemporaries, and specialized systems for graphs and arrays will be covered.

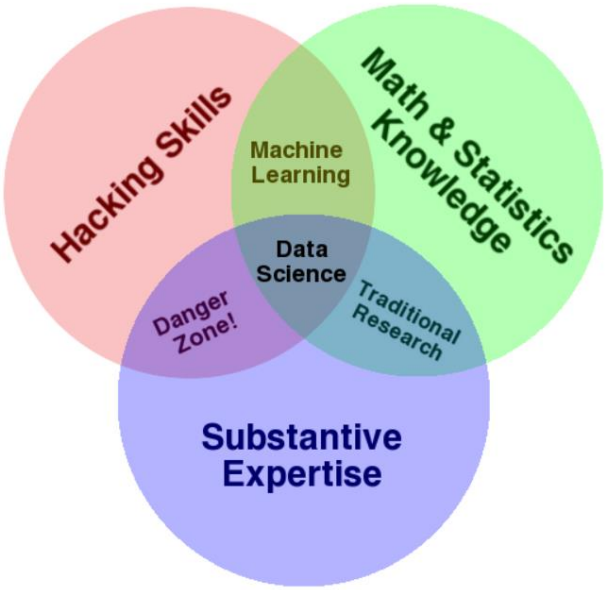
Table of Contents

1 Introduction:

The increased capacity of contemporary computers allows the gathering, storage and analysis of large amount of data which only a few years ago would have been impossible. These new data are providing large quantities of information, and enabling its interconnection using new computing methods and databases. There are many issues arising from the emergence of *big data*, from computational capacity to data manipulation techniques, all of which present challenging opportunities. Researchers and industries working in various different fields are dedicating efforts to resolve these issues. At the same time, scholars are excited by the scientific possibilities offered by *big data*, and especially the opportunities to the investigate major societal problems related to health, privacy, economics and business dynamics. Although these vast amounts of digital data are extremely informative, and their enormous possibilities have been highlighted on several occasions, issues related to their measurement, validity and reliability remain to be addressed.

Data analysis has replaced data acquisition as the bottleneck to evidence- based decision making. Extracting knowledge from large, heterogeneous, and noisy data sets requires not only powerful computing resources, but the programming abstractions to use them effectively. The abstractions that emerged in the last decade blend ideas from parallel databases, distributed systems, and programming languages to create a new class of scalable data analytics platforms that form the foundation for data science at realistic scales.

So data science refers to an emerging area of work concerned with the collection, preparation, analysis, visualization, management and preservation of large collections of information, so in this case data science in telling us about how to deal with huge instructed data that does not fit in our computers memory, and how we can build statistical models to analyze and communicate between the results by building algorithms at scale.



1

The region where the *big data* exist is somewhere between Math & statistics, Hacking, and Substantive expertise.

Any data science project has three major tasks to work with:

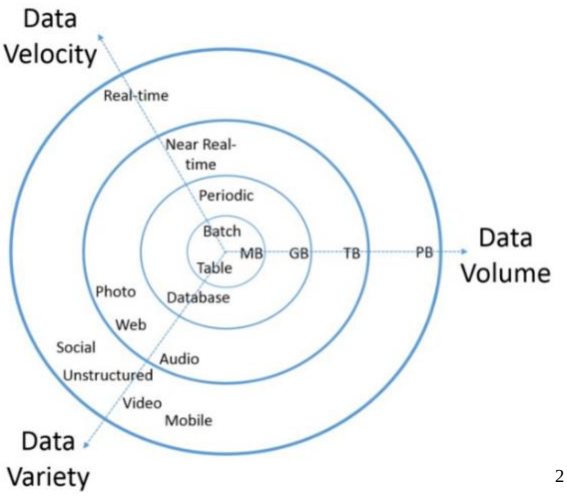
1. Preparing to run a model: where in this particular task there are many subtasks and there is a huge effort will be done during this step starting from gathering the data, to clean it, integrating, restructuring, and transforming to be all in one form, to loading, filtering, deleting, combining and merging, and at the end by verifying, extracting, shaping, to get the needed data to run the model in ignoring the unneeded data to save resources and time of processing.
2. Designing and Running a model on the prepared data
3. Interpreting the results after running the models where this step is the final end and need huge amount of analysis to extract the needed information and finding new as well

so during this report we will talk about:

- Common patterns, challenges, and approaches associated with data science projects, and what makes them different from projects in related fields, done by assignment 1.
- Also identify and use programming models associated with scalable data manipulation, including relational algebra done by assignment 2, *MapReduce*, and other data flow models done by assignment 3.
- The use of database technology adapted for large-scale analytics, including the concepts driving parallel databases, parallel query processing, and in-database analytics.
- Evaluate key-value stores and *NoSQL* systems, describe their tradeoffs with comparable systems, the details of important examples in the space, and future trends.
- Thinking in *MapReduce* to effectively write algorithms for systems including *Hadoop* and *Spark*, their limitations, design details, their relationship to databases, and their associated ecosystem of algorithms, extensions, and languages.
- Describe the landscape of specialized *Big Data* systems for graphs, arrays, and streams.

2 Characteristic of Big Data:

Big data is really critical to handle as it is emerging as one of the fastest technologies in current era. The importance of *big data* is in analytical use which can help in generating informative decision to provide better and quick service. The *big data* has three characteristics, known as data **V**olume, **V**elocity and **V**ariety, it is also known as **3V**s , which means that the size of data is large, the data is generated very fast, and the data exists in heterogeneous formats which can be among structured data, semi structured data and unstructured data captured from different sources. The common concept of **3V**s is given below:



2

2.1 Volume:

The main attraction of *big data* analytics is to process large amounts of information. The volume presents the major challenge for the traditional approaches of data analytics. It motivates the use of parallelism and distributed approach in computation. Most of the organizations and firms have large amount of data but they don't have the capacity to process it.

2.2 Velocity:

The speed at which the data generated in an organization defines the velocity of data, which is cumulatively increasing. With the increase in use of Internet with different devices the services have become faster and the services are increasingly instrumented, which gives rise in the rate of data velocity. Those who are able to quickly utilize that information, for example, by suggesting options, the company can take advantage of selling of more products. The smart phone era increases again the rate of data inflow, as consumers hold the source of data that can be in more than one form. The data should be streamed, for feasible storage space and for applications that require immediate response to the data. As with velocity of input data, velocity of output data also matters, the results may impact on decision making.

2.3 Variety:

Mostly the data is in the unordered and unstructured format which requires processing. The data may be generated from diverse sources. The data can be in the text, image, audio, video, etc. formats. None of these data are readily in the acceptable formats for integration into an application. A general characteristic of *big data* processing is to take unordered and unstructured data and extract ordered meaning, for consumption either by an individual or an application. In the course of generating processed data from the source data there can be loss of information. Based on the nature of data the storage can be fixed to make it simpler and efficient, like using relational database, XML, Graphs, etc. selecting the right approach to provide enough structure to organize data is an important part of *big data*.

3 Database:

A database (***DB***) is a collection of information that is organized so that it can be easily accessed, managed and updated[3]. Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it.

So the *DB* is used for sharing and support concurrent access by multiple readers and writers, in addition to make sure that all applications see clean, organized data, and also should be flexible by using the data in new, unanticipated ways, and the last thing is to work with datasets too large to fit in the memory, in other word that *DB* should be scaled.

So from data science point of view there should be some point that need to be taken into consideration and those are:

- How the data physically organized on disk?
- What kinds of queries are efficiently supported by this organization?
- How hard is it to update the data or add new data?
- What is the results after encountering new queries that did not anticipate? And that bring another questions that if the data should be reorganize?

Those questions can be deriving to categorize databases accordingly, during this report we are not going to describe each and every database type but we will give some brief description of the ones that we need to talk about that is related to our topic. Below are some types of *DBs*:

- Relational Databases
- Document oriented Databases
- Distributed Databases
- Key value Databases
- Centralized Databases
- Parallel Databases
- NoSQL Databases
- And many others

3.1 Document Oriented Databases:

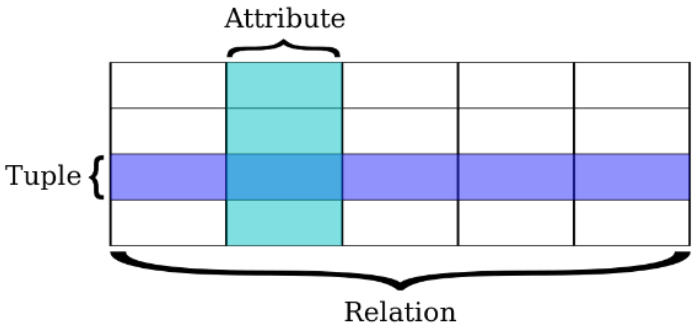
A document database is designed to store semi-structured data as documents, typically in JSON or XML format. Unlike traditional relational databases, the schema for each non-relational (*NoSQL*) document can vary, giving Developers, Database Administrators, and IT Professionals more flexibility in organizing and storing application data and reducing storage required for optional values[4].

3.2 Key Value Databases:

Key value database is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

3.3 Relational databases:

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (*DBMS*) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (*RDBMS*) performs these tasks. *DBMS* as used in this book is a general term that includes *RDBMS*[5]. This *DB* organizes data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Rows are also called records or tuples. Columns are also called attributes. Generally, each table/relation represents one "entity type". The rows represent instances of that type of entity and the columns representing values attributed to that instance.



6

Relational algebraic has two semantics and those are:

- Sets: is a collection of objects which there are no duplicates
  Sets: {a,b,c}, {a,d,e,f}, { }, . . .
- Bag: is collection of Objects which there can be a duplicate/s
  Bags: {a, a, b, c}, {b, b, b, b, b}, . . .

3.3.1 Relational Operators:

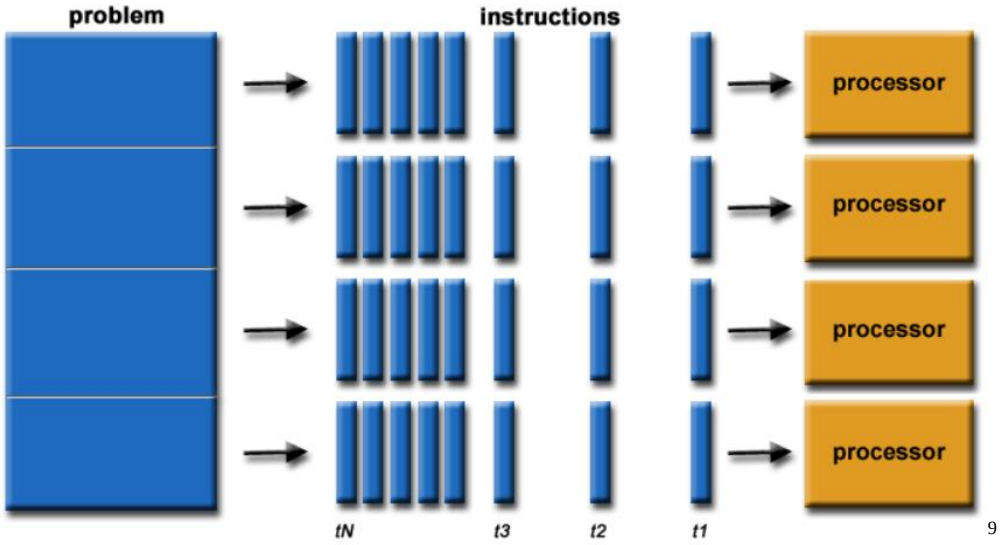
The relational operators have the property of closure that is applying operators on relations produce relations[7].

- **SELECT (s)**: also known as RESTRICT, yields values for all the rows found in a table that satisfy a given condition. SELECT yields a horizontal subset of a table.
- **PROJECT (Π)**: yields all values for selected attributes. PROJECT yields a vertical subset of a table
- **UNION (U)**: combines all rows from two or more tables, excluding duplicate rows. In order to be used in a UNION, the tables must have the same attribute characteristics, that it the attributes and their domains must be compatible. When two or more tables share the same number of columns and when their corresponding columns share the same or compatible domains, they are said to be union-compatible.
- **INTERSECT (∩)**: yields only the rows that appear in both tables. As with UNION, the tables must be union-compatible to yield valid results.
- **DIFFERENCE (-)**: yields all rows in one table that are not found in the other table, that is, it subtracts one table from the other. As with the UNION, the tables must be UNION-compatible to yield valid results.

- **PRODUCT (x)**: yields all possible pairs of rows from two tables- also known as Cartesian product.
- **JOIN (⋈)**: allows information to be combined from two or more tables. JOIN allows the use of independent tables linked by common attributes. A natural join links tables be selecting only the rows with common values in their common attributes.

4 Parallel Programming:

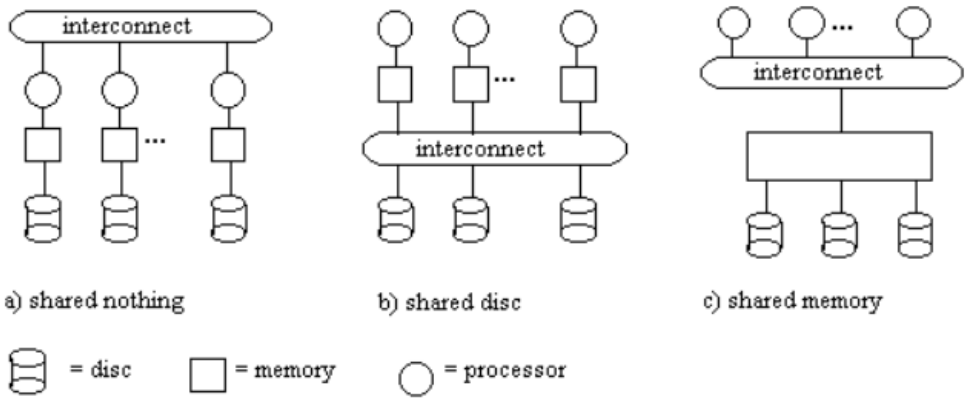
Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has long been employed in high-performance computing, but it's gaining broader interest due to the physical constraints preventing frequency scaling[8].



9

So the main tasks for parallel computing as shown in the figure above are:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



10

What is interesting us on our topic is to parallelize databases to do parallelize programming and queries on databases distributed among a cluster as shown in the above image. So as shown the first design (shared nothing) in scaled up to 1000 server, as well as the second one (shared disc), but it is so hard to program, while the third design (shared memory) are the easiest one to program but it is so expensive.

4.1 Cluster Computing:

A cluster is a large number of commodity servers, connected by high speed commodity network using parallel computing to run a big problem to distribute among the cluster to be solved faster. on the other hand cluster computing have a disadvantage that the mean time between failure is high. For instance, if each server has MTBF is 1 year in a cluster of 10,000 servers will have 1 failure/hour.

4.2 Distributed File Systems (DFS):

The Distributed File System (DFS) functions provide the ability to logically group shares on multiple servers and to transparently link shares into a single hierarchical namespace. DFS organizes shared resources on a network in a treelike structure[11].

So for a very large files (in Tera/Peta Bytes) is partitioned into chunks, where usually each one has the size of 64MegaBytes. Each chunk replicated several times on different racks in the cluster in case of failure. This have been implemented in many clusters such as: Google (*GFS*) and *Hadoop* (*HDFS*)

4.3 Distributed and parallel queries:

So for scalability, we need to increase number of working computer in cluster and that can be done by distributed query, and/or number of threads and that can be done by parallel queries.

- **Distributed Query:**
  This can happen by taking single large table and distributed across the cluster then rewriting the query as a union of sub-individual queries where workers can communicate through standard interfaces, where the final results will be send back to the main (master) server, and here is the bottle nick of this process
- **Parallel Query:**
  Here each operator is implemented with a parallel algorithm using relation algebra

5 Programming model:

During this chapter we are going to talk about *MapReduce* model, Locality sensitive hashing, Page ranking, Stream mining, and Graphs & Social Networks, and describe their concepts

5.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a *key/value* pair to generate a set of intermediate *key/value* pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system[12].

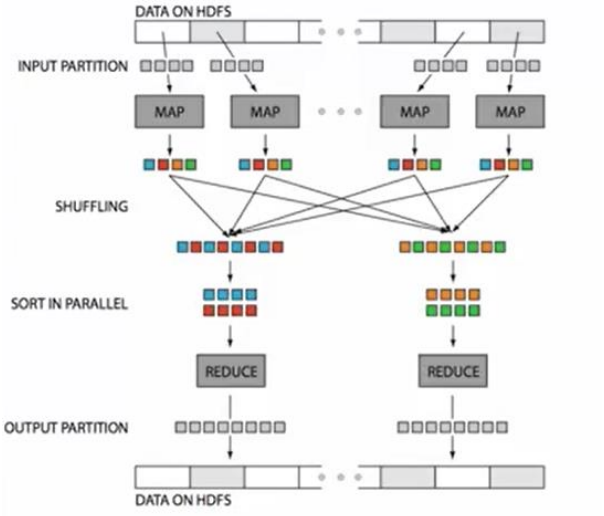
5.1.1 MapReduce implementation:

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the *MapReduce* library expresses the computation as two functions: *Map* and *Reduce*. ***Map***, written by the user, takes an input pair and produces a set of intermediate *key/value* pairs. The *MapReduce* library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

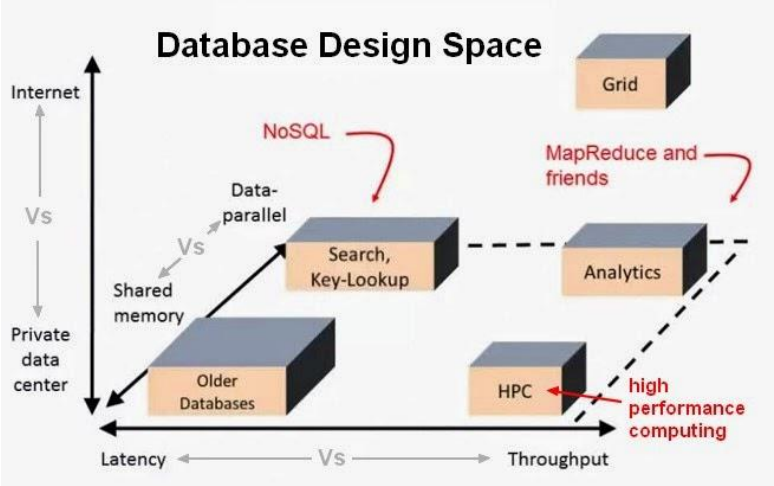
The ***Reduce*** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

To apply *MapReduce* model to a problem using a cluster:

- there should be one master node which will partition the input file into *M* splits by key and assign workers (servers) to the *M* map tasks.
- Each worker will write his own output to local disk
- The master node will assign workers to reduce tasks from the output that handled by map tasks
- All steps are shown in the figure below



So where *MapReduce* exist in Parallel computing, this question is answered by the figure below which shows that *MapReduce* model can be applied for high throughput in parallelized data to do analytics in high performance computing.



13

So as a conclusion we notice that *MapReduce* is lightweight framework can be used in parallel computing that provide:

- Automatic parallelization and distribution
- Fault-Tolerance
- Input output scheduling
- Status and monitoring

5.1.2 MapReduce libraries:

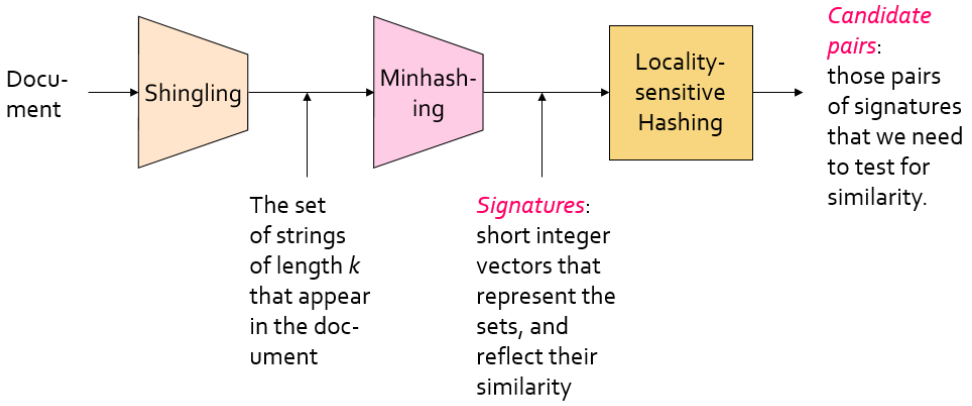
There are libraries on top of *MapReduce* can do all the needed tasks without rewriting the same function and below are some of them:

- ***Pig***: which is originally developed by Yahoo, is a high-level platform for creating programs that run on *Hadoop*. The language for this platform is called *Pig Latin* [14]. *Pig* can execute its *Hadoop* jobs in *MapReduce*, Apache Tez, or Apache Spark. *Pig Latin* abstracts the programming from the Java *MapReduce* idiom into a notation which makes *MapReduce* programming high level, similar to that of *SQL* for relational database management systems.
- **HIVE**: which is initially developed by Facebook, is a data warehouse software project built on top of *Hadoop* for providing data summarization, query and analysis [15]. Hive gives an *SQL*-like interface to query data stored in various databases and file systems that integrate with *Hadoop*. Traditional *SQL* queries must be implemented in the *MapReduce* Java API to execute *SQL* applications and queries over distributed data.
- **Impala**: which is developed by Cloudera, is a query engine that runs on *Hadoop* which brings scalable parallel database technology to *Hadoop*, enabling users to issue low-latency *SQL* queries to data stored in *HDFS* without requiring data movement or transformation. Impala is integrated with *Hadoop* to use the same file and data formats, metadata, security and resource management frameworks used by *MapReduce*, *Hive*, *Pig* and other *Hadoop* software[16].

- **HBase**: developed by Apache software foundation's Apache *Hadoop*, is an open-source, non-relational, distributed database modeled and runs on top of *HDFS*. Tables in *HBase* can serve as the input and output for *MapReduce* jobs run in *Hadoop* [17].

5.2 Locality Sensitive Hashing:

Locality sensitive hashing (*LSH*) is an algorithm for solving the approximate or exact near neighbor search in high dimensional space without a quadratic cost of finding the pairs of neighbors



18

So to do locality sensitive hashing we need to do two steps before as in the following

5.2.1 Shingling:

is a set of unique "*shingles*" (n-grams, contiguous subsequences of tokens in a document) that can be used to gauge the similarity of two documents. Which will convert a document to sets. So as an example, k= shingles=2; doc = "abcab". Set of 2-shingles = {ab, bc, ca}.

- Usually documents who are intuitively similar will have many shingles in common, and normally k is 8,9, 10 are used more often
- To save space but still make each shingle rare, we can hash them, and in that case we will call them tokens. So to represent a document by its tokens, that is, the set of hash values of its k-shingles.

5.2.2 Minhashing:

or *Jaccard* similarity is a statistic used for comparing the similarity and diversity of sample sets. The *Jaccard* coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}$$

First we need to construct small signature from large sets by building matrix which their columns correspond to the sets, and the rows correspond to the elements of the universal set from which the elements of the sets are drawn, where 1 is in the matrix only if row of the corresponding element is a member of its

column and else it will be 0, and this matrix called *characteristic matrix.* This matrix is just for visualization because they are almost always sparse

To *minhash* a set represented by a column of the characteristic matrix, pick a permutation of the rows. The *minhash* value of any column is the number of the first row, in the permuted order, in which the column has a 1.

There is a remarkable connection between *minhashing* and *Jaccard* similarity of the sets that are *minhashed*, which is the probability that the *minhash* function for a random permutation of rows produces the same value for two sets equals the *Jaccard* similarity of those sets.

Again think of a collection of sets represented by their characteristic matrix M. To represent sets, we pick at random some number n of permutations of the rows of M. Perhaps 100 permutations or several hundred permutations will do. Call the *minhash* functions determined by these permutations $h_1, h_2, \ldots, h_n$. From the column representing set S, construct the minhash signature for S, the vector $[h_1(S), h_2(S), \ldots, h_n(S)]$. We normally represent this list of hash-values as a column. Thus, we can form from matrix M a signature matrix, in which the ith column of M is replaced by the minhash signature for (the set of) the ith column.

Fortunately, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows. A hash function that maps integers 0, 1, . . . , k −1 to bucket numbers 0 through k−1 typically will map some pairs of integers to the same bucket and leave other buckets unfilled. However, the difference is unimportant as long as k is large and there are not too many collisions. We can maintain the fiction that our hash function h "permutes" row r to position h(r) in the permuted order.

Thus, instead of picking n random permutations of rows, we pick n randomly chosen hash functions $h_1, h_2, \ldots, h_n$ on the rows. We construct the signature matrix by considering each row in their given order. Let SIG(i, c) be the element of the signature matrix for the ith hash function and column *c*. Initially, set *SIG*(i, c) to ∞ for all *i* and *c*. We handle row *r* by doing the following:

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.
2. For each column c do the following:
    a. If *c* has 0 in row *r*, do nothing.
    b. However, if *c* has 1 in row *r*, then for each *i* = 1, 2, . . . , n set *SIG*(i, c) to the smaller of the current value of *SIG*(i, c) and $h_i(r)$.

The cost of *minhashing* is proportional to the number of rows and this will lead to save a lot of time, but in case that we have for example 1000 rows have 0 in one column, we will get no *minhash* value. But this can be improved by dividing the rows into bands (*k*) and in that case we will need to compute only $(1/k)^{th}$ of the number of hash values per row if we will use the original scheme.

5.2.2.1 Example:

Let us reconsider the characteristic matrix, which we reproduce with some additional data as shown below.

ROW	S_1	S_2	S_3	S_4	$x+1\ mod 5$	$3x+1\ mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

We have chosen two hash functions: $h_1(x) = x+1 \ mod \ 5$ and $h_2(x) = 3x+1 \ mod \ 5$. The values of these two functions applied to the row numbers are given in the last two columns. Notice that these simple hash functions are true permutations of the rows, but a true permutation is only possible because the number of rows, 5, is a prime. In general, there will be collisions, where two rows get the same hash value. Now, let us simulate the algorithm for computing the signature matrix. Initially, this matrix consists of all ∞'s:

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

First, we consider row 0. We see that the values of $h_1(0)$ and $h_2(0)$ are both 1. The row numbered 0 has 1's in the columns for sets S_1 and S_4, so only these columns of the signature matrix can change. As 1 is less than ∞, we do in fact change both values in the columns for S_1 and S_4. The current estimate of the signature matrix is thus:

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Now, we move to the row numbered 1. This row has 1 only in S_3, and its hash values are $h_1(1) = 2$ and $h_2(1) = 4$. Thus, we set *SIG*(1, 3) to 2 and *SIG*(2, 3) to 4. All other signature entries remain as they are because their columns have 0 in the row numbered 1. The new signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

The row in the original characteristic matrix numbered 2 has 1's in the columns for S_2 and S_4, and its hash values are $h_1(2) = 3$ and $h_2(2) = 2$. We could change the values in the signature for S_4, but the values in this column of the signature matrix, [1, 1], are each less than the corresponding hash values [3, 2]. However, since the column for S_2 still has ∞'s, we replace it by [3, 2], resulting in:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

Next comes the row numbered 3 in characteristic matrix. Here, all columns but S_2 have 1, and the hash values are $h_1(3) = 4$ and $h_2(3) = 0$. The value 4 for h_1 exceeds what is already in the signature matrix for all the columns, so we shall not change any values in the first row of the signature matrix. However, the value 0 for h_2 is less than what is already present, so we lower *SIG*(2, 1), *SIG*(2, 3) and *SIG*(2, 4) to 0. Note that we cannot lower *SIG*(2, 2) because the column for S_2 in characteristic matrix has 0 in the row we are currently considering. The resulting signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	1

Finally, consider the row of characteristic matrix numbered 4. $h_1(4) = 0$ and $h_2(4) = 3$. Since row 4 has 1 only in the column for S_3, we only compare the current signature column for that set, [2, 0] with the hash values [0, 3]. Since 0 < 2, we change *SIG*(1, 3) to 0, but since 3 > 0 we do not change *SIG*(2, 3). The final signature matrix is:

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	1

We can estimate the *Jaccard* similarities of the underlying sets from this signature matrix. Notice that columns 1 and 4 are identical, so we guess that SIM(S_1, S_4) = 1.0. If we look at characteristic matrix, we see that the true *Jaccard* similarity of S_1 and S_4 is 2/3. Remember that the fraction of rows that agree in the signature matrix is only an estimate of the true *Jaccard* similarity, and this example is much too small for the law of large numbers to assure that the estimates are close. For additional examples, the signature columns for S_1 and S_3 agree in half the rows (true similarity 1/4), while the signatures of S_1 and S_2 estimate 0 as their *Jaccard* similarity (the correct value).

5.2.3 LSH:

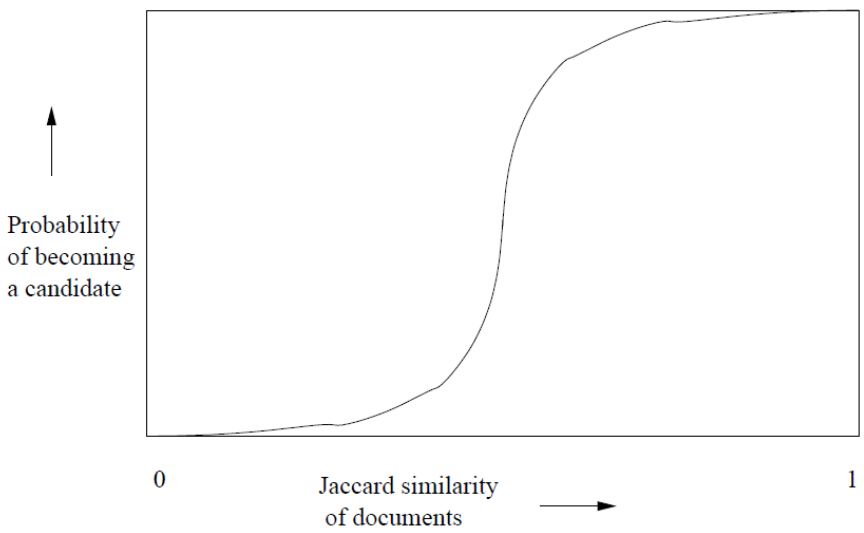
One general approach to *LSH* is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair that hashed to the same bucket for any of the *hashings* to be a candidate pair. We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked. Those dissimilar pairs that do hash to the same bucket are false positives; we hope these will be only a small fraction of all pairs. We also hope that most of the truly similar pairs will hash to the same bucket under at least one of the hash functions. Those that do not are false negatives; we hope these will be only a small fraction of the truly similar pairs.

If we have *minhash* signatures for the items, an effective way to choose the *hashings* is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

Suppose we use *b* bands of *r* rows each, and suppose that a particular pair of documents have *Jaccard* similarity *s*. so the probability the *minhash* signatures for these documents agree in any one particular row of the signature matrix is *s*. We can calculate the probability that these documents (or rather their signatures) become a candidate pair as follows:

1. The probability that the signatures agree in all rows of one particular band is s^r
2. The probability that the signatures disagree in at least one row of a particular band is $1 - s^r$
3. The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$
4. The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$

It may not be obvious, but regardless of the chosen constants *b* and *r*, this function has the form of an S-curve, as shown below.



The threshold, that is, the value of similarity *s* at which the probability of becoming a candidate is 1/2, is a function of *b* and *r*. The threshold is roughly where the rise is the steepest, and for large *b* and *r* there we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want.

We can now give an approach to finding the set of candidate pairs for similar documents and then discovering the truly similar documents among them. It must be emphasized that this approach can produce false negatives – pairs of similar documents that are not identified as such because they never become a candidate pair. There will also be false positives – candidate pairs that are evaluated, but are found not to be sufficiently similar.

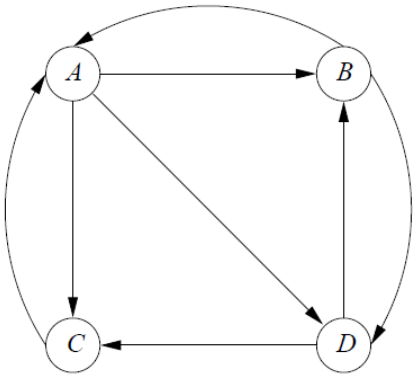
1. Pick a value of *k* and construct from each document the set of *k-shingles*. Optionally, hash the *k-shingles* to shorter bucket numbers.
2. Sort the document-shingle pairs to order them by *shingle*.
3. Pick a length n for the *minhash* signatures. Feed the sorted list to the algorithm of *minhashing* signature to compute the *minhash* signatures for all the documents.
4. Choose a threshold *t* that defines how similar documents have to be in order for them to be regarded as a desired "similar pair." Pick a number of bands *b* and a number of rows *r* such that *br* = *n*, and the threshold t is approximately $(1/b)^{1/r}$. If avoidance of false negatives is important, you may wish to select *b* and *r* to produce a threshold lower than *t*; if speed is important and you wish to limit false positives, select *b* and *r* to produce a higher threshold.
5. Construct candidate pairs by applying the *LSH* technique.
6. Examine each candidate pair's signatures and determine whether the fraction of components in which they agree is at least *t*.
7. Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are truly similar, rather than documents that, by luck, had similar signatures.

5.3 Page Ranking:

One of the biggest changes in our lives in the decade following the turn of the century was the availability of efficient and accurate Web search, through search engines such as Google. While Google was not the first search engine, it was the first able to defeat the spammers who had made search almost useless. Moreover, the innovation provided by Google was a nontrivial technological advance, called *PageRank*[19].

PageRank is a function that assigns a real number to each page in the Web (or at least to that portion of the Web that has been crawled and its links discovered). The intent is that the higher the *PageRank* of a page, the more *important* it is. There is not one fixed algorithm for assignment of *PageRank*, and in fact variations on the basic idea can alter the relative *PageRank* of any two pages.

The figure below is an example of a tiny version of the Web, where there are only four pages. Page A has links to each of the other three pages; page B has links to A and D only; page C has a link only to A, and page D has links to B and C only.



Suppose a random surfer starts at page A. There are links to B, C, and D, so this surfer will next be at each of those pages with probability 1/3, and has zero probability of being at A. A random surfer at B has, at the next step, probability 1/2 of being at A, 1/2 of being at D, and 0 of being at B or C.

In general, we can define the transition matrix of the Web to describe what happens to random surfers after one step. This matrix *M* has n rows and columns, if there are *n* pages. The element m_{ij} in row *i* and column *j* has value *1/k* if page *j* has *k* arcs out, and one of them is to page *i*. Otherwise, $m_{ij} = 0$. So the transition matrix for the web in our example is:

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

If you look at the 4-node "Web", we might think that the way to solve the equation $v = Mv$, where v is Eigen vector, is by Gaussian elimination. Indeed, in that example, we argued what the limit would be essentially by doing so. However, in realistic examples, where there are tens or hundreds of billions of nodes, Gaussian elimination is not feasible. The reason is that Gaussian elimination takes time that is cubic in the number of equations. Thus, the only way to solve equations on this scale is to iterate as we have suggested. Even that iteration is quadratic at each round, but we can speed it up by taking advantage of the fact that the matrix *M* is very sparse; there are on average about ten links per page.

Back to our example if we start with v_0 in all probability equal to ¼ this will lead by multiplying at each step by *M* to:

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}, \begin{bmatrix}9/24\\5/24\\5/24\\5/24\end{bmatrix}, \begin{bmatrix}15/48\\11/48\\11/48\\11/48\end{bmatrix}, \ldots\ldots, \begin{bmatrix}3/9\\2/9\\2/9\\2/9\end{bmatrix}$$

The first row of *M* tells us that the probability of A must be *3/2* the other probabilities, so the limit has the probability of A equal to *3/9*, or *1/3*, while the probability for the other three nodes is *2/9*.

The page with no link out is called a dead end that lead to transition matrix of the web is no longer stochastic, so there are two approaches to deal with dead ends:

1. We can drop the dead ends from the graph, and also drop their incoming arcs. Doing so may create more dead ends, which also have to be dropped, recursively. However, eventually we wind up with a strongly-connected component, none of whose nodes are dead ends.
2. We can modify the process by which random surfers are assumed to move about the Web. This method, which we refer to as "taxation," also solves the problem of spider traps.

If we use the first approach, recursive deletion of dead ends, then we solve the remaining graph *G* by whatever means are appropriate, including the taxation method if there might be spider traps in G. Then, we restore the graph, but keep the *PageRank* values for the nodes of *G*. Nodes not in *G*, but with predecessors all in *G* can have their *PageRank* computed by summing, over all predecessors *p*, the *PageRank* of p divided by the number of successors of *p* in the full graph. Now there may be other nodes, not in *G*, that have the *PageRank* of all their predecessors computed. These may have their own *PageRank* computed by the same process. Eventually, all nodes outside *G* will have their *PageRank* computed; they can surely be computed in the order opposite to that in which they were deleted.

To avoid the problem of spider traps, we modify the calculation of *PageRank* by allowing each random surfer a small probability of teleporting to a random page, rather than following an out-link from their current page. The iterative step, where we compute a new vector estimate of *PageRanks v′* from the current *PageRank* estimate *v* and the transition matrix *M* is

$$v' = \beta M v + (1-\beta)e/n$$

Where β is chosen constant, usually in a range 0.8 to 0.9, and e is a vector of all 1's with the appropriate number of components, and *n* is the number of nodes in the web graph.

PageRank is similar to the equation we used for general *PageRank*. The only difference is how we add the new surfers. Suppose *S* is a set of integers consisting of the row/column numbers for the pages we have identified as belonging to a certain topic (called the teleport set). Let *eS* be a vector that has 1 in the components in *S* and 0 in other components. Then the topic-sensitive *PageRank* for *S* is the limit of the iteration

$$v' = \beta M v + (1-\beta)eS/|S|$$

Here, as usual, *M* is the transition matrix of the Web, and |*S*| is the size of set *S*.

Notes:

- For Web-sized graphs, it may not be feasible to store the entire *PageRank* estimate vector in the main memory of one machine. Thus, we can break the vector into *k* segments and break the transition matrix into k^2 squares, called blocks, assigning each square to one machine. The vector segments are each sent to *k* machines, so there is a small additional cost in replicating the vector.

- When we divide a transition matrix into square blocks, the columns are divided into *k* segments. To represent a segment of a column, nothing is needed if there are no nonzero entries in that segment. However, if there are one or more nonzero entries, then we need to represent the segment of the column by the total number of nonzero entries in the column (so we can tell what value the nonzero entries have) followed by a list of the rows with nonzero entries[20].
- We can parallelize the process to calculate *PageRank* by using simple *MapReduce* algorithm to perform matrix-vector multiplication, but since matrix is sparse, it is better to treat it as a relational join. another approach is to use many jobs, each to multiply a row of matrix blocks by the entire Eigen vector

5.3.1 Hubs and Authorities:

While *PageRank* gives a one-dimensional view of the importance of pages, an algorithm called *HITS* tries to measure two different aspects of importance. Authorities are those pages that contain valuable information. Hubs are pages that, while they do not themselves contain the information, link to places where the information can be found[21].

To formalize the above intuition, we shall assign two scores to each Web page. One score represents the *hubbiness* of a page – that is, the degree to which it is a good hub, and the second score represents the degree to which the page is a good authority. Assuming that pages are enumerated, we represent these scores by vectors *h* and *a*. The ith component of *h* gives the *hubbiness* of the ith page, and the ith component of *a* gives the authority of the same page.

While importance is divided among the successors of a page, as expressed by the transition matrix of the Web, the normal way to describe the computation of *hubbiness* and authority is to add the authority of successors to estimate *hubbiness* and to add *hubbiness* of predecessors to estimate authority. If that is all we did, then the *hubbiness* and authority values would typically grow beyond bounds. Thus, we normally scale the values of the vectors *h* and *a* so that the largest component is 1. An alternative is to scale so that the sum of components is 1.

To describe the iterative computation of *h* and *a* formally, we use the link matrix of the Web, *L*. If we have *n* pages, then *L* is an $n \times n$ matrix, and $L_{ij} = 1$ if there is a link from page *i* to page *j*, and $L_{ij} = 0$ if not. We shall also have need for L^T, the transpose of *L*. That is, $L^T_{ij} = 1$ if there is a link from page *j* to page *i*, and $L^T_{ij} = 0$ otherwise. Notice that L^T is similar to the matrix *M* that we used for *PageRank*, but where L^T has 1, *M* has a fraction – 1 divided by the number of out-links from the page represented by that column.

5.3.2 Link Spam:

Link spam is to fool the *PageRank* algorithm; unscrupulous actors have created spam farms. These are collections of pages whose purpose is to concentrate high *PageRank* on a particular target page. Where the structure of spam farm typically consists of a target page and very many supporting pages. The target page links to all the supporting pages, and the supporting pages link only to the target page. In addition, it is essential that some links from outside the spam farm be created. For example, the spammer might introduce links to their target page by writing comments in other people's blogs or discussion groups[22].

One way to ameliorate the effect of link spam is to compute a topic-sensitive *PageRank* called *TrustRank*, where the teleport set is a collection of trusted pages. And this can be used to identify spam farms, we can compute both the conventional *PageRank* and the *TrustRank* for all pages. Those pages that have much lower *TrustRank* than *PageRank* are likely to be part of a spam farm.

5.4 Stream Mining:

The algorithms for processing streams each involve summarization of the stream in some way. We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate

most of the "undesirable" elements. We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Stream data model is a model that assumes data arrives at a processing engine at a rate that makes it infeasible to store everything in active storage. One strategy to dealing with streams is to maintain summaries of the streams, sufficient to answer the expected queries about the data. A second approach is to maintain a sliding window of the most recently arrived data.

5.4.1 Filtering and Bloom filters:

We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated, then the selection is easy to do.

An empty Bloom filter is a bit array of *m* bits, all set to 0. There must also be *k* different hash functions defined, each of which maps or hashes some set element to one of the *m* array positions, generating a uniform random distribution. Typically, *k* is a constant, much smaller than *m*, which is proportional to the number of elements to be added; the precise choice of *k* and the constant of proportionality of *m* are determined by the intended false positive rate of the filter[23].

5.4.2 Sampling a stream:

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

So if we want to test our sample we will choose 10% of the length of whole stream, but her we sampled based on the position in the stream, rather than the value of the stream element, so to do better we need to hash search queries to 10 buckets (0,1,….,9) and sample all search queries that hash to 0 bucket. In case sample size is limited then we hash to a larger number of buckets. This technique called sampling by value. Another technique called sampling by key-value pairs which generalizes to any form of data that we can see as tuples (*k*,*v*) where *k* is key and *v* is value, so we hash keys to some number of buckets, and in this case sample consists of all key-value pairs with a key that goes into one of the selected buckets.

We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past. So we will use the Flajolet Martin Algorithm[24]

5.4.2.1 Flajolet Martin Algorithm:

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see. As we see more different hash-values, it becomes more likely that one of these values will be "unusual." The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist[25].

Whenever we apply a hash function *h* to a stream element *a*, the bit string *h(a)* will end in some number of 0's, possibly none. Call this number the tail length for *a* and *h*. Let *R* be the maximum tail length of any *a* seen so far in the stream. Then we shall use estimate 2^R for the number of distinct elements seen in the stream.
This estimate makes intuitive sense. The probability that a given stream element *a* has *h(a)* ending in at least *r* 0's is 2^{-r}. Suppose there are *m* distinct elements in the stream. Then the probability that none of them has tail length at least *r* is $(1-2^{-r})^m$. This sort of expression should be familiar by now.

We can rewrite it as $((1-2^{-r})^{2^r})^{m2^{-r}}$. Assuming r is reasonably large, the inner expression is of the form $(1-\epsilon)^{1/\epsilon}$, which is approximately *1/e*. Thus, the probability of not finding a stream element with as many as r 0's at the end of its hash value is $e^{-m2^{-r}}$. We can conclude:

- If m is much larger than 2^r, then the probability that we shall find a tail of length at least r approaches 1.
- If m is much less than 2^r, then the probability of finding a tail length at least r approaches 0.

We conclude from these two points that the proposed estimate of m, which is 2^R (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

5.5 Graphs and Social Networks:

Graphs is a set of vertices *v* and edges *e* which each edge is a pair of two vertices and considered to be directed or undirected. Web, Internet, Social networks, and communication logs can be considered as graphs. The most important definitions that define a graph is the number of edges while usually it is much higher that number of vertices and more interesting for analytics, another concept [26] to know are:

- **in-degree**(*v*) which is number of edges going in to specific *v*
- **out-degree**(*v*) is the number of edges going out from *v*
- **Connectivity Coefficient**: the minimum number of vertices you need to remove that will disconnect the graph
- **Closeness Centrality**: average length of all its shortest paths
- **Betweeness centrality of a vertex**: the fraction of all shortest paths that pass through *v*
- **Degree centrality:** is the degree of a vertex over the number of edges
- **PageRank:** as we explain it before in section 5.3
- **Minimum spanning tree:** which is the smallest subset of edges that weakly connect the graph
- **Maximum flow:** subgraph that maximize the flow between sources and sinks (destination)

We need a way to represent a graph while graphs become bigger more than before such as Brain graph that can be defined as 100 billion vertices, with 100 trillion edges, there are multiple ways to represent graphs like that and do analysis on it like using *MapReduce* algorithm for *PageRank* definition.

Social networks are naturally modeled as graphs, which we sometimes refer to as a social graph. The entities are the nodes, and an edge connects two nodes if the nodes are related by the relationship that characterizes the network. If there is a degree associated with the relationship, this degree is represented by labeling the edges. Often, social graphs are undirected, as for the Facebook friends graph. But they can be directed graphs, as for example the graphs of followers on Twitter.

5.5.1 Betweenness:

To understand more about graphs we need to define the *betweenness* of an edge (a, b) to be the number of pairs of nodes x and y such that the edge (a, b) lies on the shortest path between x and y. we can find *betweenness* by performing *Girvan-Newman algorithm*[27], and that will be by the following:

1. Perform a breadth-first search from each node of the graph
2. Label nodes top-down to count the number of the shortest paths from the root to that node
3. Label both nodes and edges bottom-up with the sum, over nodes N at or below, of the fraction of the shortest paths from the root to N going through that node or edge
4. Betweenness of an edge is half the sum of labels of that edge, starting with each node as root

5.5.2 Triangles:

Another interesting thing in graph is finding *triangles* because triangles can measure the maturity of a community, so we observe that undirected graph have N nodes and M edges where $N \leq M \leq N^2$ and we can do that by considering all N 3 sets of nodes and check if there are edges connecting them all, but this approach is so complex $O(N^3)$. We can improve this algorithm by considering all edges *e* and all nodes *u* and see if both ends of *e* have edges to *u*, and here the complexity of this algorithm is *O(MN)*. To improve more we need to use the concept *Heavy hitter* which is a node with degree at least $\sqrt{M}$. So a *heavy hitter triangle* is one whose three nodes are all *heavy hitters* and complexity of it is $O(M^{1.5})$

5.5.3 Neighborhoods:

Neighborhood of a node *u* at distance *d* is the set of all nodes *v* such that there is a path of length at most *d* from *u* to *v* and it is denoted by *n(u,d)*, so the sizes of neighborhoods of small distance measure the influence of node in a social network.

In this concept we can also apply Flagolet-Martin algorithm to estimate the number of distinct elements in the union of several sets.

6 NoSQL:

A *NoSQL* is a database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases [28]. *NoSQL* databases are increasingly used in *big data* and real-time web applications [29].

Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases) [30], and finer control over availability. The data structures used by *NoSQL* databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in *NoSQL*. The particular suitability of a given *NoSQL* database depends on the problem it must solve [31].

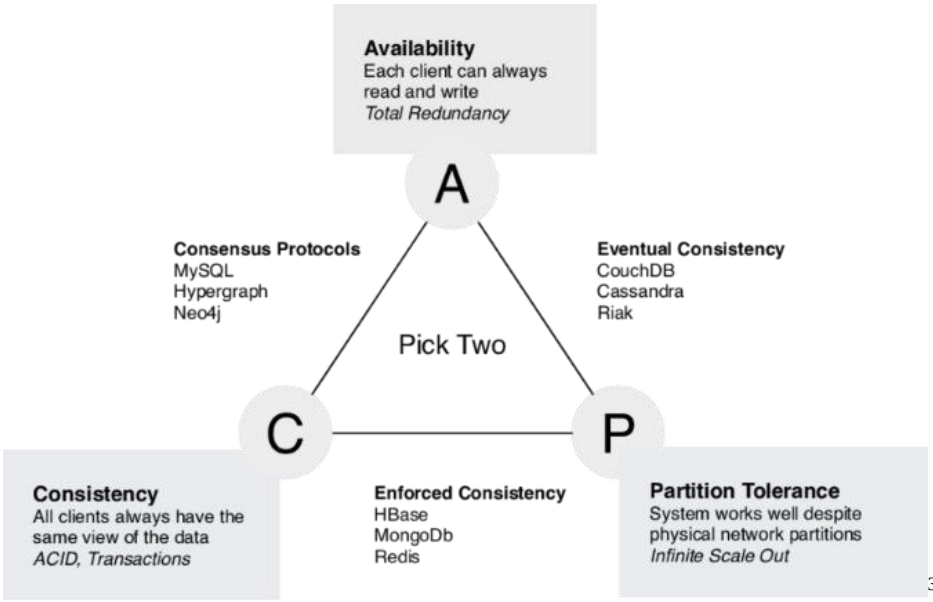
Things to take it in our consideration is to ensure high availability of data by replicate the chunks of data in the cluster in case of failure, in addition we want also to support in case of any update on the data, and this can be done using multi algorithms such as two phase commit, where if there is an update data should be ensuring that receiver servers should be available for any update and communicate to make sure that they are ready and the new status of data is successfully updated. In addition, there is another way such as *PAXOS*, and others.

NoSQL systems generally have six key features and they are:

- The ability to horizontally scale simple operation throughput over many servers
- The ability to replicate and distribute (partition) data over many servers
- A simple call level interface or protocol (in contrast to *SQL* binding)
- A weaker concurrency models that ACID transactions of most relational *RDBMS*
- Efficient use of distributed indexes and RAM for data storage
- The ability to dynamically add new attributes to data records

6.1 CAP theorem:

CAP theorem it comes from (Consistency, availability, and partitioning) as shown in the figure below



[32]

- **Consistency**: is making sure that all applications see the same data, this condition may not fit *NoSQL* systems
- **Availability**: is to make sure if there will be any action in the system in case of failure of data updates
- **Partitioning**: is to divide the data among cluster, but here in case of failure between two sections in the system to communicate, does it make any forward progress or it stops, so in case it stops we should sacrifice availability, otherwise we should sacrifice consistency

6.2 NoSQL Systems:

We will provide a short summary of the most important features of different *NoSQL* systems in order to examine performance advantages or disadvantages compared to traditional relational database systems, below is a small table of comparing multiple systems of *NoSQL* and related systems as well.

Year	*System*	*Scale to 1000s*	*Primary index*	*Secondary indexes*	*Transactions*	*Joins/Analytics*	*Integrity Constraints*	*Views*	*Language/ Algebra*	*Data Model*	*Label*	*Notes*
1971	RDBMS	o	✓	✓	✓	✓	✓	✓	✓	tables	sql-like	
2003	memcached	✓	✓	o	o	o	o	o	o	key-val	nosql	
2004	MapReduce	✓	o	o	o	✓	o	o	o	key-val	batch	
2005	CouchDB	✓	✓	✓	record	MapReduce (MR)	o	✓	o	document	nosql	
2006	Hbase	✓	✓	✓	record	compat. w/MR	/	o	o	ext. record	nosql	
2007	MongoDB	✓	✓	✓	EC / record	o	o	o	o	document	nosql	
2007	Dynamo	✓	✓	o	o	o	o	o	o	ext. record	nosql	
2008	Pig	✓	o	o	o	✓	/	o	✓	tables	sql-like	
2008	Cassandra	✓	✓	✓	EC / record	o	✓	✓	o	key-val	nosql	
2009	Riak	✓	✓	✓	EC / record	MR	o			key-val	nosql	
2010	Dremel	✓	o	o	o	/	✓	o	✓	tables	sql-like	
2011	Spark/Shark	✓	o	o	o	✓	✓	o	✓	tables	sql-like	
2012	Spanner	✓	✓	✓	✓	?	✓	✓	✓	tables	sql-like	
2ß13	Impala	✓	o	o	O	✓	✓		✓	tables	sql-like	

In the next sessions we will discuss some of the mentioned systems

6.2.1 Memcached:

Memcached (Main memory caching service) is a general-purpose distributed memory caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source (such as a database or *API*) must be read [33].

Memcached's APIs provide a very large hash table distributed across multiple machines. When the table is full, subsequent inserts cause older data to be purged in least recently used order. Applications using *Memcached* typically layer requests and additions into RAM before falling back on a slower backing store, such as a database.

6.2.2 Dynamo:

Dynamo is the name given to a set of techniques that when taken together can form a highly available key-value structured storage system or a distributed data store [34]. It has properties of both databases and distributed hash tables (*DHTs*).

Dynamo used multiple principles and those are:

- **Incremental scalability**: *Dynamo* should be able to scale out one storage host (or "node") at a time, with minimal impact on both operators of the system and the system itself.
- **Symmetry**: Every node in *Dynamo* should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities.
- **Decentralization**: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control.
- **Heterogeneity**: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

6.2.3 CoughDB:

CouchDB is open source database software that focuses on ease of use and having a scalable architecture. It has a document-oriented *NoSQL* database architecture.

Unlike a relational database, a *CouchDB* database does not store data and relationships in tables. Instead, each database is a collection of independent documents. Each document maintains its own data and self-contained schema. An application may access multiple databases, such as one stored on a user's mobile phone and another on a server. Document metadata contains revision information, making it possible to merge any differences that may have occurred while the databases were disconnected[35].

6.2.4 BigTable(HBase):

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on *Bigtable*, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, *Bigtable* has successfully provided a flexible, high-performance[36].

HBase is an open-source, non-relational, distributed database modeled after Google's *Bigtable*. *HBase* features compression, in-memory operation, and Bloom filters on a per-column basis. Tables in *HBase* can serve as the input and output for *MapReduce* jobs run in *Hadoop*. *HBase* is a column-oriented key-value data store and has been idolized widely because of its lineage with *Hadoop* and *HDFS* (*Hadoop* distributed file system). *HBase* runs on top of *HDFS* and is well-suited for faster read and write operations on large datasets with high throughput and low input/output latency.

6.3 SQL-like systems with scalability:

These systems mostly have a lot of features that looks like *SQL* and *RDBMS* where in the other hand it can scale:

6.3.1 Pig:

is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of *Pig* programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets. At the present time, *Pig's* infrastructure layer consists of a compiler that produces sequences of *MapReduce* programs, for which large-scale parallel implementations already exist (e.g., the *Hadoop* subproject). *Pig's* language layer currently consists of a textual language called *Pig Latin*, which has the following key properties[37]:

- **Ease of programming**: It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
- **Optimization opportunities:** The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
- **Extensibility:** Users can create their own functions to do special-purpose processing.

As mentioned before *Pig* is built on top of *MapReduce* but it used a high level language which start to optimize the work on it to reduce writing in the code to provide the needed results using *MapReduce* algorithm.

6.3.2 Spark:

Spark was developed in response to limitations in the *MapReduce* cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: *MapReduce* programs read input data from disk, map a function across the data, *reduce* the results of the *map*, and store reduction results on disk. Spark's function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory [38].

Spark facilitates the implementation of both iterative algorithms, that visit their data set multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. The latency of such applications may be reduced by several orders of magnitude compared to a *MapReduce* implementation (as was common in *Hadoop* stacks) [39]. *Spark* requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster), *Hadoop*. For distributed storage, Spark can interface with a wide variety, including *Hadoop* Distributed File System (HDFS), and others.

7 Assignments:

This section contains 3 main assignments, each part contains sub tasks which worked with *python*, and *liteSQL*

7.1 Assignment 1:

As a matter of fact, to understand the concept of *big data*, the best way to deal with is to practice, and accordingly, I worked with twitter to apply some fundamentals procedures and calculations for social measurements, where millions of people voluntarily express opinions across any topic imaginable, and this data is incredibly valuable for both research and business. So during this assignment we did the following:

1. Learning python by Google Python class
2. Accessing twitter **A**pplication **P**rogramming **I**nterface (API) using *python*:

   So to access twitter live stream, we need to have twitter account and create new app on it using this link by following the instruction in that link. After we need to get "**Consumer Key (API Key)**", and "**Consumer Secret (API Secret)**", and "**Access token**" and also "**Access token secret**", and need to fill these keys in the code of extract twitter live data code. After we need to run the code for a couple of minutes so we can extract live stream data, and during this task we need to submit the first 20 lines (tweets) from our stream data.
3. Estimate the public perception of a particular term or phrase:

   Here multiple tasks are requested and they are as the following:

   i. We will compute the sentiment of each tweet based on the sentiment scores of the terms in the tweet. The sentiment of a tweet is equivalent to the sum of the sentiment scores for each term in the tweet. We are provided with text file which contains a list of pre-computed sentiment scores. Each line in the file contains a word or phrase followed by sentiment score. Each word or phrase that is found in a tweet but not found in this file should be given a sentiment score of 0. The data in the tweet file we generated in the second task is represented as *JSON*, which stands for **J**ava**S**cript **O**bject **N**otation. It is a simple format for representing nested structures of data --- lists of lists of dictionaries of lists of ..... items. Each line of output file from task 2 represents a streaming message. Most, but not all, will be tweets. It is straightforward to convert a *JSON* string into a *Python* data structure; there is a library to do so called json, we need to import it. So to load *json* data we will use *json.loads* for each line that return python data structure (dictionary). So at the end we need to provide a sentiment score for each tweet in the output file from task 2 which, taking in our consideration that we will apply scores just for English tweets where the others we do not have pre-computed sentiment score for words in other languages, the problem that we face that data are really so messy because it is real world data, so we reduce the work by ignoring punctuations such as (? : ! . ; " @ ') from the text of the tweets.

   ii. During this task need to provide a score for the terms that does not appear in the provided pre-computed sentiment score file, so we create four functions and those are the following:

      1. Creating dictionary from provided pre-computed sentiment word score file to load in the memory
      2. Reading text section in each tweet in a string array
      3. Deriving sentiment from each tweet by summing up sentiments of individual word

4. Derive terms of sentiments that are defined or undefined in the provided pre-computed sentiment word score file.

iii. In this task we will compute term frequency histogram of the live stream data that we get it in task 2 which can be calculated as the following

[#No of occurrences of the term in all tweets] / [#No of occurrences of all terms in all tweets]

The output should be the word followed by frequency in the whole file
We create a function to read the tweets and append in the string array of tweets then create *term_list* that provided the words that occurred in the input file, and *total_list* which the last is dictionary correspond to the number of occurrence that term over number of total terms. We run in each tweet, each word, checking if they exist in the term_list or not and add it if not then calculating the appearance of it and saving in the *total_list*. At the end the output will be as word followed by (appearance of this word / total_number_of_words).

4. Analyze the relationship between location and mood based on a sample of twitter data:
   During this task we had two scripts which provide the happiest state in US and the top ten hash tags which we will compute from our output file and below is the description for each one:
   i. For the happiest state we define dictionary list of states in US with their abbreviations that abbreviation exist in twitter data as *json* model.
   So first we need to check the place with country code that is US and full name in the place is not none while, sometimes the user does not want to provide information for their own privacy, so we can deal with those tweets only, then we need to for each tweet in each state we need to calculate the sentiment from pre-computed sentiment word score file that is provided and add the value of the score of each tweet that is related to state in list which at the end we will calculate the score for each state divided by (number of tweets) for each and choosing the maximum value between them, this will return the happiest state to us. There is another way to define tweet state and this after checking the country, twitter provide the longitude and latitude of the tweets so we can define area for each state and compare the longitude and latitude for each tweet if it is one of the states, so we can add it to the list and complete the same work accordingly, but this need pre-defined data for each area (longitude and latitude) that unfortunately, we do not have it.
   ii. For top ten hashtags we define two functions one for extracting the hashtags that can be found in *json* model of twitter in [“entities”][“hashtags”] and the other function is to find the top ten by listing them in frequency list that contains tuples of [hashtag, count] for each and sorting them in reverse way then print out the first ten hashtags with their counts.
5. More information and details about the assignment can be found in the following link
6. All the source codes are provided for this assignment in Appendix A and additional files are provided as attachment with the project

7.2 Assignment 2:

During this assignment we will cover dealing with relational algebraic databases by using *SQLite* , provided with database file "reuters.db" which consist of single table frequency (*docid*, *term*, *count*) where *docid* is a document identifier corresponding to a particular file of *text*, term is an English word, and *count* is the number of the occurrences of the term within the document indicated by *docid*. This assignment contains 3 main problems and some sub problems described as below:

- **Problem 1**: Inspecting the Reuters Dataset; Basic Relational Algebra:
  - **select**: Query that is equivalent to the following relational algebra expression σdocid=10398_txt_earn(frequency)

    here we need to know how to use select and run query to find count of occurrence of one document identifier that is pre-defines in the question

  - **select project**: Write a *SQL* statement that is equivalent to the following relational algebra
    πterm(σdocid=10398_txt_earn and count=1(frequency))

    we need to find special document identifier with count of 1 in the database file and return how many records with these values.

  - **union**: *SQL* statement that is equivalent to the following relational algebra expression. (Hint: you can use the UNION keyword in *SQL*)
    πterm(σdocid=10398_txt_earn and count=1(frequency)) U πterm(σdocid=925_txt_trade and count=1(frequency))

    we need to find number of document identifier with count 1 in the database combine or sum the result with another count of document identifier with count 1 in the same database

  - **count**: *SQL* statement to count the number of documents containing the word "parliament"

  - **big documents**: *SQL* statement to find all documents that have more than 300 total terms, including duplicate terms.
    By *grouping* summation of the count by "sum(count)" we put a condition that use have statement that sum more than 300

  - **two words**: *SQL* statement to count the number of unique documents that contain both the word 'transactions' and the word 'world'.
    In this task we use *inner join* to find the both terms that we defined

- **Problem 2**: Matrix Multiplication in *SQL*
  - **multiply**: Express A x B as a *SQL* query
    Systems designed to efficiently support sparse matrices look a lot like databases: They represent each cell as a record (*i*,*j*,*value*). The benefit is that you only need one record for every non-zero element of a matrix. Since you can represent a sparse matrix as a

table, it's reasonable to consider whether you can express matrix multiplication as a SQL query and whether it makes sense to do so.
Within *matrix.db*, there are two matrices A and B represented as follows:

A(row_num, col_num, value)
B(row_num, col_num, value)

The matrix A and matrix B are both square matrices with 5 rows and 5 columns each, where columns are sorted in an ascending way starting from row number after column number for both tables (matrixes A, and B) so we need to sum the multiplication of the values of in A for the rows and B for the columns that have the same value and making sure that column of A for that value is equal to row of B to get the right multiplications, then to aggregate the values by the same representation of rows of A and columns of B to get the value of the needed element

- **Problem 3**: Working with a Term-Document Matrix
  - **similarity matrix**: Query to compute the similarity matrix DDT.
    The reuters dataset can be considered a term-document matrix, which is an important representation for text analytics. Each row of the matrix is a document vector, with one column for every term in the entire corpus. Naturally, some documents may not contain a given term, so this matrix is rather sparse. The value in each cell of the matrix is the term frequency.
    So to compute the similarity we can multiply the matrix with its own transpose $S = DD^T$, and we will have an (unnormalized) measure of similarity. The result is a square document-document matrix, where each cell represents the similarity. Here, similarity is pretty simple: if two documents both contain a term, then the score goes up by the product of the two term frequencies. This score is equivalent to the dot product of the two document vectors.
    So the solution is, we define two virtual tables A and B which they are the same as frequency from *reuters.db* and we join them together where term of A should be equal to the term of B and we sum the multiplication of the count for these terms for pre-defined identifiers for A and B which bill be aggregated as *docid* of A and B

  - **keyword search**: Best matching document to the keyword query "Washington taxes treasury".
    Here we need to do the same as the task before but have specific words in the terms column as mentioned before which we will define as union keywords and have count 1 with document identifier is *q*. the solution is by providing those specific conditions first and filter the table and provided as view table called keywords then find the similarity according to the same query used in the example before and set it in order to extract the highest similarity value as the question

- All *SQL* queries are provided for this assignment in Appendix B and additional files are provided as attachment with the project.

7.3 Assignment 3:

This assignment contains 6 problems, where in each we design and implement *MapReduce* algorithm, for variety of common data processing tasks:

- **Problem 1**: An Inverted index. Given a set of documents, an inverted index is a dictionary where each word is associated with a list of the document identifiers in which that word appears, where:
  Mapper Input:
  The input is a 2 element list: [document_id, text], where document_id is a string representing a document identifier and text is a string representing the text of the document. The document text may have words in upper or lower case and may contain punctuation. We should treat each token as if it was a valid word; that is, you can just use value.split() to tokenize the string.
  Reducer Output:
  The output should be a (word, document ID list) tuple where word is a String and document ID list is a list of Strings.

- **Problem 2**: Implementing a relational join query as a *MapReduce*
  MapReduce query should produce the same result as the *SQL* query below, executed against an appropriate database.
  *SELECT **
  FROM Orders, LineItem
  WHERE Order.order_id = LineItem.order_id
  We will can consider the two input tables, Order and LineItem, as one big concatenated bag of records that will be processed by the map function record by record. So the implementation is:
  Map Input
  Each input record is a list of strings representing a tuple in the database. Each list element corresponds to a different attribute of the table
  The first item (index 0) in each record is a string that identifies the table the record originates from. This field has two possible values:
  - "line_item" indicates that the record is a line item.
  - "order" indicates that the record is an order.

  The second element (index 1) in each record is the order_id.
  LineItem records have 17 attributes including the identifier string.
  Order records have 10 elements including the identifier string.
  Reduce Output
  The output should be a joined record: a single list of length 27 that contains the attributes from the order record followed by the fields from the line item record. Each list element should be a string.

- **Problem 3**: A *MapReduce* algorithm to count the number of friends for each person in a simple social network data set consist of set of key-value pairs (person, friend).
  Map Input
  Each input record is a 2 element list [personA, personB] where personA is a string representing the name of a person and personB is a string representing the name of one of personA's friends. Note that it may or may not be the case that the personA is a friend of personB.
  Reduce Output
  The output should be a pair (person, friend_count) where person is a string and friend_count is an integer indicating the number of friends associated with person.

- **Problem 4**: Implementation of a *MapReduce* algorithm to generate a list of all non-symmetric friend relationships.
  Map Input
  Each input record is a 2 element list [personA, personB] where personA is a string representing the name of a person and personB is a string representing the name of one of personA's friends. Note that it may or may not be the case that the personA is a friend of personB.
  Reduce Output
  The output should be all pairs (friend, person) such that (person, friend) appears in the dataset but (friend, person) does not.
- **Problem 5**: by given a set of key-value pairs where each key is sequence id and each value is a string of nucleotides, e.g., GCTTCCGAAATGCTCGAA....
  The task is to write *MapReduce* query to remove the last 10 characters from each string of nucleotides, then remove any duplicates generated.
  Map Input
  Each input record is a 2 element list [sequence id, nucleotides] where sequence id is a string representing a unique identifier for the sequence and nucleotides is a string representing a sequence of nucleotides
  Reduce Output
  The output from the reduce function should be the unique trimmed nucleotide strings.

- **Problem 6**: A *MapReduce* algorithm to compute the matrix multiplication A x B for sparse matrixes format where each record is of the form (i ,j ,value) that are integers.
  Map Input
  The input to the map function will be a row of a matrix represented as a list. Each list will be of the form [matrix, i, j, value] where matrix is a string and i, j, and value are integers.
  The first item, matrix, is a string that identifies which matrix the record originates from. This field has two possible values: "a" indicates that the record is from matrix A and "b" indicates that the record is from matrix B
  Reduce Output
  The output from the reduce function will also be a row of the result matrix represented as a tuple. Each tuple will be of the form (i, j, value) where each element is an integer.

- All source codes are provided for this assignment in Appendix C and additional files are provided as attachment with the project.

8 Conclusion:

We have entered an era of *Big Data*. Through better analysis of the large volumes of data that are becoming available, there is the potential for making faster advances in many scientific disciplines and improving the profitability and success of many enterprises. However, many technical challenges described in this paper must be addressed before this potential can be realized fully. The challenges include not just the obvious issues of scale, but also heterogeneity, lack of structure, error-handling, privacy, timeliness, provenance, and visualization, at all stages of the analysis pipeline from data acquisition to result interpretation. These technical challenges are common across a large variety of application domains, and therefore not cost-effective to address in the context of one domain alone. Furthermore, these challenges will require transformative solutions, and will not be addressed naturally by the next generation of industrial products. We must support and encourage fundamental research towards addressing these technical challenges if we are to achieve the promised benefits of *Big Data*.

9 Appendix A:

Extract live twitter data:

```
import oauth2 as oauth
import urllib2 as urllib

api_key = "W9Qt28JxRtOPKxnqsDyfv8iFk"
api_secret = "XQWKpvTjoUKGDvJDIAActAzthLU7ls0ywjjzXNAPc1m7cW1Pta"
access_token_key = "406303992-JJgg8J2pcmjmCAq3OMAEc76DWH9Gs0DA3orV5NYO"
access_token_secret = "JUa9O5eXKPqjLORfpqMtFYUCFOmJr5sGbUcxZqRnhhcoT"

_debug = 0

oauth_token    = oauth.Token(key=access_token_key, secret=access_token_secret)
oauth_consumer = oauth.Consumer(key=api_key, secret=api_secret)

signature_method_hmac_sha1 = oauth.SignatureMethod_HMAC_SHA1()

http_method = "GET"

http_handler  = urllib.HTTPHandler(debuglevel=_debug)
https_handler = urllib.HTTPSHandler(debuglevel=_debug)

'''
Construct, sign, and open a twitter request
using the hard-coded credentials above.
'''
def twitterreq(url, method, parameters):
  req = oauth.Request.from_consumer_and_token(oauth_consumer,
                                              token=oauth_token,
                                              http_method=http_method,
                                              http_url=url,
                                              parameters=parameters)

  req.sign_request(signature_method_hmac_sha1, oauth_consumer, oauth_token)

  headers = req.to_header()

  if http_method == "POST":
    encoded_post_data = req.to_postdata()
  else:
    encoded_post_data = None
    url = req.to_url()

  opener = urllib.OpenerDirector()
  opener.add_handler(http_handler)
  opener.add_handler(https_handler)

  response = opener.open(url, encoded_post_data)

  return response

def fetchsamples():
  url = "https://stream.twitter.com/1.1/statuses/sample.json"
  parameters = []
  response = twitterreq(url, "GET", parameters)
  for line in response:
    print line.strip()

if __name__ == '__main__':
  fetchsamples()
```

Derive the sentiment of each tweet:

```
import sys
import json

def main():
    sent_file = open(sys.argv[1])
    tweet_file = open(sys.argv[2])

    # Read sentiment and create dic from it
    scores = {}
    for line in sent_file:
      term, score  = line.split("\t")
      scores[term] = int(score)
    sent_file.close()

    # read tweet_file
    tweet_data = []
    for line in tweet_file:
        response = json.loads(line)
        if "text" in response.keys():
            tweet_data.append(response["text"])

    # encode the tweet, delete unneccessery characters, lower the case for each word and calculate the score
    for tweet in tweet_data:
        sum = 0
        encoded_tweet = tweet.encode('utf-8')
        words = encoded_tweet.split()

                for word in words:
            word = word.rstrip('?:!.,;"!@')
                    word = word.lower()
                    if word in scores:
                sum = sum + scores[word]

        print '%d' % sum

if __name__ == '__main__':
    main()
```

Derive the sentiment of new terms

```
import sys
import json

#Read sentiment and create dic from it
def dictFromSentFile(sent_file):
    scores = {}
    for line in sent_file:
      term, score  = line.split("\t")
      scores[term] = int(score)
    sent_file.close()
    return scores

# read tweet_file
def readTweetFile (tweet_file):
    tweet_data = []
    for line in tweet_file:
        response = json.loads(line)
        if "text" in response.keys():
                tweet_data.append(response["text"])
```

```
    return tweet_data

# Derive sentiment from each tweet by summing up sentiments of individual words
def tweetSentiment (tweet_data, scores):
    sentiments = []
    for tweet in tweet_data:
        sum = 0.0
        encoded_tweet = tweet.encode('utf-8')
        words = encoded_tweet.split()
            for word in words:
          word = word.rstrip('?:!.,;"!@')
                word = word.lower()
                if word in scores:
            sum = sum + scores[word]
            sentiments.append(sum)
    return sentiments

# derive terms of sentiments
def termSentiment (tweet_data, scores, sentiments):
            index = 0
            repeatance = {}
            for tweet in tweet_data:
                        words = tweet.encode('utf-8').split()
                        for word in words:
                            repeatance [word] = 0
                        for word in words:
                                    repeatance[word] = repeatance[word] + 1
                                    if word not in scores:
                                                scores[word] = sentiments[index]
                                    else:
                                                scores[word] = (scores[word] + sentiments[index]) / repeatance[word]  #
average
                                    print word + " %.3f" %scores[word]
                        index = index + 1
            return scores

def main():
    sent_file = open(sys.argv[1])
    tweet_file = open(sys.argv[2])
    scores = dictFromSentFile(sent_file)
    tweet_data = readTweetFile(tweet_file)
    tweet_sentiments = tweetSentiment(tweet_data, scores)
    termSentiment(tweet_data, scores, tweet_sentiments)

if __name__ == '__main__':
    main()
```

Compute term frequency

```
import sys
import json

# read tweet_file
def readTweetFile (tweet_file):
    tweet_data = []
    for line in tweet_file:
        response = json.loads(line)
        if "text" in response.keys():
            tweet_data.append(response["text"])
    return tweet_data

def main():
```

```
    tweet_file = open(sys.argv[1])
    tweets = readTweetFile(tweet_file)
    term_list = []
    total_list = {}#dict of term and corresponding the number of occurance that term/ number of total terms
    for tweet in tweets:
            words = tweet.encode('utf-8').split()
            for word in words:
               word = word.rstrip('?:!.,;"!@')
               word = word.lower()
               term_list.append(word)

    for word in term_list:
            if word in total_list:
                    total_list[word] = total_list[word] + 1
            else:
                    total_list[word] = 1
    total_number = len(total_list)

    for word in total_list:
       total_list[word] = "%.3f" %(float(total_list[word])/total_number)
       print word + " " + total_list[word]

if __name__ == '__main__':
    main()
```

Which State is happiest

```
import sys
import json

#Read sentiment and create dic from it
def dictFromSentFile(sent_file):
    scores = {}
    for line in sent_file:
      term, score  = line.split("\t")
      scores[term] = int(score)
    sent_file.close()
    return scores

# read tweet_file
def readTweetFile (tweet_file):
    tweet_data = []
    for line in tweet_file:
       tweet_data.append(json.loads(line))
    return tweet_data

def main():
    sent_file = open(sys.argv[1])
    tweet_file = open(sys.argv[2])
    scores = dictFromSentFile(sent_file)
    tweets = readTweetFile(tweet_file)
    state_dict =
{"Alabama":"AL","Alaska":"AK","Arizona":"AZ","Arkansas":"AR","California":"CA","Colorado":"CO","Connecticu
t":"CT","Delaware":"DE","Florida":"FL","Georgia":"GA","Hawaii":"HI","Idaho":"ID","Illinois":"IL","Indiana":"IN","
Iowa":"IA","Kansas":"KS","Kentucky":"KY","Louisiana":"LA","Maine":"ME","Maryland":"MD","Massachusetts":"
MA","Michigan":"MI","Minnesota":"MN","Mississippi":"MS","Missouri":"MO","Montana":"MT","Nebraska":"NE","
Nevada":"NV","NewHampshire":"NH","NewJersey":"NJ","NewMexico":"NM","NewYork":"NY","NorthCarolina":"
NC","NorthDakota":"ND","Ohio":"OH","Oklahoma":"OK","Oregon":"OR","Pennsylvania":"PA","RhodeIsland":"RI",
"SouthCarolina":"SC","SouthDakota":"SD","Tennessee":"TN","Texas":"TX","Utah":"UT","Vermont":"VT","Virginia"
:"VA","Washington":"WA","WestVirginia":"WV","Wisconsin":"WI","Wyoming":"WY"}
    state_list = {}
```

```
    max_score = 0
    happiest_state = ""

    for tweet in range(len(tweets)):
        if 'place' in tweets[tweet].keys() and tweets[tweet]["place"] is not None and
tweets[tweet]["place"]["country_code"]=='US' and tweets[tweet]["place"]["full_name"] is not None:
                tweet_word = tweets[tweet]["text"].encode('utf-8').lower().split()
                tweet_country = tweets[tweet]["place"]["country_code"]

                        if tweets[tweet]["place"]["full_name"].split(", ")[1] == "USA":
                                tweet_state_full_name = tweets[tweet]["place"]["full_name"].split(", ")[0]
                                tweet_state = state_dict[tweet_state_full_name]
                        else:
                                tweet_state = tweets[tweet]["place"]["full_name"].split(", ")[1]
                        #print tweet_word

                        if len(tweet_state) == 2 and tweet_country == 'US':
                                #print tweet_state
                    sent_score = 0
                        for word in tweet_word:
                        if word in scores.keys():
                            sent_score = sent_score + float(scores[word])
                        else:
                            sent_score = sent_score

                    if tweet_state in state_list.keys():
                        state_list[tweet_state].append(sent_score)
                    else:
                        state_list[tweet_state] = []
                        state_list[tweet_state].append(sent_score)
                                #print state_list[tweet_state]

    for state in state_list.keys():
            state_score = 0
            state_list_score = state_list[state]

            for score in state_list_score:
                state_score = state_score + float(score)

            state_score = state_score/len(state_list[state])

            if happiest_state == "" or state_score > max_score:
                max_score = state_score
                happiest_state = state

    print happiest_state, max_score

if __name__ == '__main__':
    main()
```

Top ten hash tags

```
import sys
import json
import operator
from operator import itemgetter

# read tweet_file
def readTweetFile (tweet_file):
    tweet_file = open(sys.argv[1])
    tweet_data = []
    for line in tweet_file:
```

```
        tweet_data.append(json.loads(line))
    return tweet_data

#Extract hash tags
def extract_htags(tweets):
        htags = []
        for tweet in tweets:
                #Ensure that there is an entities element to extract.
                if "entities" in tweet.keys() and "hashtags" in tweet["entities"]:
                        for htag in tweet["entities"]["hashtags"]:
                                unicode_tag = htag["text"].encode('utf-8')
                                htags.append(unicode_tag)
        return htags

#Count top 10 hash tags
def top_ten(htags):
        freq = []
        for htag in htags:
                tup = [htag,htags.count(htag)]
                #print tup
                if tup not in freq :
                    freq.append(tup)
        freq_sorted = sorted(freq, key=itemgetter(1), reverse=True)

        for i in range(0,10):
                print freq_sorted[i][0] + " " + str(float(freq_sorted[i][1]))

def main():
    freq = []
    tweets = readTweetFile(sys.argv[1])
    htags = extract_htags(tweets)
    top_ten(htags)

if __name__ == '__main__':
        main()
```

10 Appendix B:

Problem 1:

```
/* A select σdocid=10398_txt_earn(frequency) */
select count(*)
from (
select *
from frequency A
where A.docid = '10398_txt_earn'
);

/* B πterm(σdocid=10398_txt_earn and count=1(frequency)) */
select count(*)
from (
select term
from frequency B
where B.docid = '10398_txt_earn' and B.count = 1
);

/* C πterm(σdocid=10398_txt_earn and count=1(frequency)) U πterm(σdocid=925_txt_trade and count=1(frequency))
*/
select count(*)
from (
select term
from frequency C1
where C1.docid = '10398_txt_earn'and C1.count=1
union select term
from frequency C2
where C2.docid = '925_txt_trade' and C2.count=1
);

/* D count the number of documents containing the word "parliament" */
select count(*)
from (
select *
from frequency D
where D.term = 'parliament'
);

/* E more than 300 total terms */
select count(*)
from (
select docid, sum(count)
from frequency E
group by E.docid
having sum(E.count) > 300
);

/* F unique documents that contain both the word 'transactions' and the word 'world' */
select count(*)
from (
select F1.docid
from frequency F1
inner join frequency F2 on F1.docid = F2.docid
where F1.term = 'transactions' and F2.term = 'world'
);
```

Problem 2:

```
/* Matrix Multiplication */
select A.row_num, B.col_num, sum(A.value * B.value)
from A, B
where A.col_num = B.row_num
--and a.row_num = 2 and b.col_num = 3
group by A.row_num, B.col_num;
```

Problem 3:

```
/* H Similarity Matrix */
select H
from (
select A.docid, B.docid, sum(A.count * B.count) as H
from frequency as A
join frequency as B
on A.term = B.term
where A.docid < B.docid
and A.docid = '10080_txt_crude'
and B.docid = '17035_txt_earn'
group by A.docid, B.docid
);

/* I Keyword search */
create view if not exists keywords as
select *
from frequency
union select 'q' as docid, 'washington' as term, 1 as count
union select 'q' as docid, 'taxes' as term, 1 as count
union select 'q' as docid, 'treasury' as term, 1 as count;

select A.docid, B.docid, sum(A.count * B.count) as similarity
from keywords A join keywords B on A.term = B.term
where A.docid = 'q' and B.docid != 'q'
group by A.docid, B.docid
order by similarity desc;
```

11 Appendix C:

Problem1:

```
import MapReduce
import sys

mr = MapReduce.MapReduce()

def mapper(record):
        key = record[0]
        value = record[1]

        for word in value.split():
                mr.emit_intermediate(word,key)

def reducer(key, list_of_values):
        result=[]
        for document_ID in list_of_values:
                if document_ID not in result:
                        result.append(document_ID)
        mr.emit((key, result))

inputdata = open(sys.argv[1])
mr.execute(inputdata,mapper,reducer)
```

Problem2:

```
import sys
import MapReduce

mr=MapReduce.MapReduce()

def mapper(record):
        key = record[1]
        mr.emit_intermediate(key, record)

def reducer(key, list_of_values):
        lines = []
        order = []
        for v in list_of_values:
                if v[0] == "order":
                        order = v
                else:
                        lines.append(v)
        for line in lines:
                mr.emit(order + line)

inputData = open(sys.argv[1])
mr.execute(inputData,mapper,reducer)
```

Problem3:

```
import sys
import MapReduce

mr = MapReduce.MapReduce()

def mapper(records):
        key = records[0]
        mr.emit_intermediate(key,1)

def reducer(key,list_of_values):
        count = 0
        for value in list_of_values:
                count+=value
        mr.emit((key,count))


inputData = open(sys.argv[1])
mr.execute(inputData,mapper,reducer)
```

Problem4:

```
import sys
import MapReduce

mr=MapReduce.MapReduce()
        mr.emit( (pairs[1], pairs[0]) )

def mapper(record):
    # key: document identifier
    # value: document contents
    person = record[0]
    friend = record[1]
    mr.emit_intermediate((person,friend), 1)
    mr.emit_intermediate((friend,person), 1)

def reducer(key, list_of_values):
    # key: word
    # value: list of occurrence counts

   if len(list_of_values) < 2:
      mr.emit(key)


inputData = open(sys.argv[1])
mr.execute(inputData,mapper,reducer)
```

Problem5:

```
import sys
import MapReduce

mr = MapReduce.MapReduce()

def mapper(records):
        nucleotides = records[1][:-10]
        mr.emit_intermediate(nucleotides, 1)

def reducer(key,list_of_values):
         mr.emit(key)
```

```
inputData = open(sys.argv[1])
mr.execute(inputData,mapper,reducer)
```

Problem6:

```
import sys
import MapReduce

mr = MapReduce.MapReduce()

#Matrix Dimensions are 5x5 for A and B so the result will be also 5x5
Dim = 5

def mapper(records):
    key = records[0]
    i = records[1]
    j = records[2]
    value = records[3]

    if key == "a":
    mr.emit_intermediate(key, [i,j,value])
    elif key == "b":
    mr.emit_intermediate(key, [j,i,value])
    else:
    print "Error."

def reducer(key, list_of_values):
    A = {}
    B = {}
    result = 0
    if key == "a":
        for a in mr.intermediate["a"]:
            A[(a[0], a[1])] = a[2]
        for b in mr.intermediate["b"]:
            B[(b[0], b[1])] = b[2]
        # fill in zeros
        for i in range(0,Dim):
            for j in range(0,Dim):
                k = (i,j)
                if k not in A.keys():
                    A[k] = 0
                if k not in B.keys():
                    B[k] = 0
        # now do the multiply.
        for i in range(0,Dim):
            for j in range(0,Dim):
                result = 0
                for k in range(0,Dim):
                    result += A[(i,k)] * B[(j,k)]
                mr.emit((i,j,result))


inputData = open(sys.argv[1])
mr.execute(inputData,mapper,reducer)
```

12 References:

[1] http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram
[2] https://www.linguamatics.com/blog/big-data-real-world-data-where-does-text-analytics-fit
[3] http://searchsqlserver.techtarget.com/definition/database
[4] https://aws.amazon.com/nosql/document/
[5] https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html
[6] https://upload.wikimedia.org/wikipedia/commons/thumb/7/7c/Relational_database_terms.svg/350px-Relational_database_terms.svg.png
[7] http://cs.tsu.edu/ghemri/cs346/classnotes/relat_alg1.pdf
[8] Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.:Benjamin/Cummings. ISBN 0-8053-0177-1.
[9] https://computing.llnl.gov/tutorials/parallel_comp/images/parallelProblem.gif
[10] https://www.techscribe.co.uk/thesis/images/fig3_1.gif
[11] https://msdn.microsoft.com/en-us/library/bb524800(v=vs.85).aspx
[12] Jeffrey Dean, Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Google, 2004
[13] https://gerardnico.com/wiki/data/database/software
[14] http://pig.apache.org/
[15] Venner Jason (2009), Pro Hadoop, ISBN 978-1-4302-1942-2
[16] http://impala.apache.org/
[17] http://hbase.apache.org/poweredbyhbase.html
[18] https://stackoverflow.com/questions/12952729/how-to-understand-locality-sensitive-hashing
[19] S. Brin and L. Page, "Anatomy of a large-scale hyper-textual web search engine," Proc. 7th Intl. World-Wide-Web Conference, pp. 107–117, 1998.
[20] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Weiner, "Graph structure in the web," Computer Networks 33:1–6, pp. 309–320, 2000.
[21] J.M. Kleinberg, "Authoritative sources in a hyperlinked environment," J. ACM 46:5, pp. 604–632, 1999.
[22] Z. Gy¨ongi, H. Garcia-Molina, and J. Pedersen, "Combating link spam with trustrank," Proc. 30th Intl. Conf. on Very Large Databases, pp. 576–587, 2004.
[23] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Comm. ACM 13:7, pp. 422–426, 1970.
[24] P.B. Gibbons, "Distinct sampling for highly-accurate answers to distinct values queries and event reports," Intl. Conf. on Very Large Databases, pp. 541–550, 2001.
[25] Flajolet, Philippe; Fusy, Éric; Gandouet, Olivier; Meunier, Frédéric (2007). "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm"
[26] Richard J. Trudeau, Introduction to graph theory, Dover publications (1973); ISBN: 0486678709
[27] Lj. Despalatović, T. Vojković, D.Vukičević ; Community structure in networks: improving the Girvan-Newman algorithm; University of Split (2014)
[28] http://nosql-database.org/
[29] https://db-engines.com/en/blog_post/23
[30] http://www.leavcom.com/pdf/NoSQL.pdf
[31] https://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html
[32] https://www.researchgate.net/figure/CAP-Theorem_221462089
[33] https://www.memcached.org/
[34] Decandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. (2007). "Dynamo". Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles; ISBN 9781595935915
[35] Joab Jackson (2010); CouchDB NoSQL database ready for production use; article by PCWorld
[36] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber; Bigtable: A distributed storage system for structured data; Google (2006)
[37] https://pig.apache.org/
[38] Zaharia Matei; Chowdhury Mosharaf; Das Tathagata; Dave Ankur; Justin Ma; McCauley Murphy; J. Michael; Shenker Scott; Stoica Ion. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing; (2012) University of California

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica; Spark: Cluster computing with working sets; (2011) university of California