**Abstract**

We present a novel indexing and querying technique for large-scale datasets in Resource Description Framework (RDF) format. This technique is developed for a set of connected components that are strategically partitioned from large RDF datasets. Our approach has two major components.

First is the index-building step where we intelligently process the partitioned components of a large dataset and then construct a counting bloom filter on each of them. We then store these bloom filters in a tree structure for fast retrieval.

The second component is querying which consists of processing a query graph. We first construct a counting bloom on it and then run the query on only a few selected connected components using the bloom comparison technique.

Finally we provide an in-depth performance evaluation to illustrate that our approach works very well compared to one of the best contemporary tools used to query and index RDF data.

**Introduction**

The Semi Structured Data has been a very happening field in the database community in recent times. One of the challenging problems that are being studied is to manage semi structure data. As the name suggests semi-structured data is a form of structured data that does not confirm with the formal structure of data models associated with the relational databases. One of the widely represented semi structure formats is RDF (Resource Description Framework) format. RDF is a popular language for representing data on the Web.

The essence of RDF lies in the notion of representing any fact as subject, predicate, and object. Formally, RDF represents resources as a directed, labeled graph where a pair of adjacent nodes denotes two things and the directed, labeled edge represents their relationship. The source node denotes the subject; the sink node denotes the object; and the edge label is the predicate (or property). This “subject-predicate-object” relationship is commonly referred to as an RDF triple. SPARQL is a popular query language for RDF graphs. Using SPARQL, complex graph pattern queries can be expressed on individual RDF graphs as well as across multiple RDF graphs. In recent years, the RDF data model has become increasingly important in domain-specific applications and the WWW. Through RDF technologies, one can reason over semantic data, which is highly appealing in domains such as healthcare, defense and intelligence, biopharmaceuticals, and so forth. With the rapidly growing size of RDF datasets (e.g., DBPedia, Billion Triples Challenge), there is a pressing need for efficient RDF processing tools.

Focus of the Thesis

The focus of the thesis is to design, develop and evaluate an RDF indexing and querying technique with the following components -

1. Indexing -
2. Connected Component Processing: Every triple in a connected component is processed by generating six other arrangements for it by swapping uri's with variables.
3. Data Signature Generation: The RDF triples from the above step are converted into unsigned poly integers. This is called as a signature.
4. Data Bloom Counters: Each signature is then stored in bloom counters. Storing signatures in bloom counters helps us use perform quick bitwise operations. This we plan to use at the time of querying to quickly eliminate non-potential signatures from querying without actually going through the entire data set, thus saving a lot of time.
5. Signature Tree Generation: All the Bloom counters for each signature from the above step is then stored in a special signature tree for quick traversals and retrievals.
6. Querying -
7. Query Signature Generation: Just like the Data Signature, Query Signature is generated by converting each query triple into an unsigned poly integer.
8. Query Bloom Counters: A bloom counter is constructed for the query signature.
9. Candidate Matches: By comparing the query bloom counter with data set bloom counters, potential candidates are identified which may contain the results.
10. Query Execution: In the final step, rdf3x tool is run only on the candidate signatures to produce the results.
11. Performance Evaluation -
12. An extensive performance evaluation for this approach has been done using some of the popular RDF datasets. We choose LUBM, which is a synthetic RDF dataset to conduct our experiments. The queries for LUBM data can be classified into two sets - Queries with large number of joins and queries with less number of joins. We show that our approach outclasses the existing methods in the case of queries with large number of joins while in the case of queries with lesser joins, our approach is quite comparable with the existing methods.

The rest of the document is organized as follows. Chapter 2 discusses the background and motivation while Chapter 3 discusses Bloom Counter Generation and Comparison. Chapter 4 discusses our architecture. Chapter 5 discusses our Index Generation. Chapter 5 discusses our Query Processing. Chapter 7 discusses our evaluation. Finally, we conclude in chapter 8.

**Background and Motivations**

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs. These tools either store and process RDF in main-memory, use an RDBMS, or a native RDF database. The popular approach has been  to use relational database systems for storing, indexing, and querying RDF. Abadi et al. proposed a vertical partitioning approach and leveraged a column-oriented DBMS for achieving an order of magnitude performance improvement over previous techniques [16]. RDF-3X [17] and Hexastore [18] demonstrated that storing RDF data in a single triples table and building exhaustive indexes on the six per    mutations of (s, p, o) triples can significantly outperform the vertical partitioning approach [16] and also support a larger class of RDF queries efficiently. Recently, BitMat [19] was proposed to overcome the overhead of large intermediate join results in RDF-3X and Hexastore when queries contain low selectivity triple patterns. (Low selectivity implies large result set size.) There are some RDF stores that operate in shared-nothing clusters (e.g., YARS2 [20], 4store [21], Clustered TBD [22])  by hashing triples/quadruples and distributing them on different  nodes in the cluster. Parallel query processing is performed. The scalability of these approaches has been demonstrated on  small sized clusters. Weaver et al. [23] have studied RDF query processing on supercomputers. More recently, tools for data intensive computing such as Apache Hadoop and Pig have been used for query processing and analytics over RDF data [24], [25], [26]. These approaches are more suitable for  batch processing of queries. A few researchers have focused  on parallel RDF reasoning [27], [28]. More recently, Huang et al. , developed parallel RDF query processing techniques for large RDF graphs [29].

Architecture

Querying

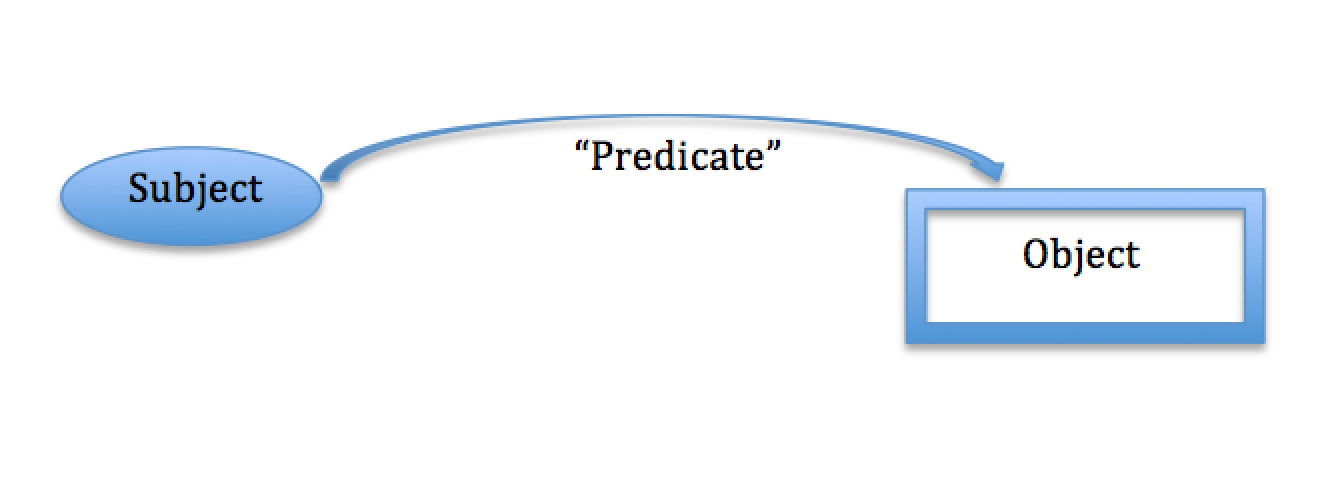
Sample Queries

Implementation and Performance Evaluation

Conclusion and Future Work

**Chapter ? RDF**

RDF stands for Resource Description Framework. It is an idea to represent statements regarding a resource as an expression involving subject, predicate and object. In RDF terminology, these expressions are often referred to as RDF triples. In an RDF triple, a subject is the resource involved in the statement, predicate is a property about the resource and object is a defining characteristic of the source connected via a predicate. In most cases, predicates represent a relationship between the subject and object. Usually, a dot ‘.’ Is used at the end of a triple to mark that the expression is complete.



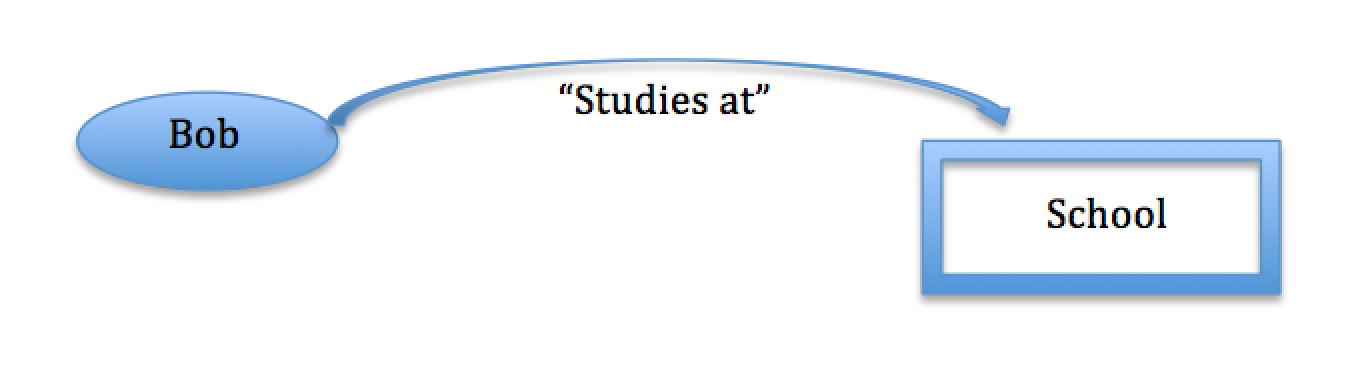
Example - **Bob studies at school**

In the above statement,

Subject/Resource is “Bob”,

Predicate/property is “studies at” and,

Object is “school”



**More complex examples**

RDF graphs can be used to illustrate complex graphs too. There could be graphs that describe millions of subjects and have millions of properties and objects.

 It is also very typical that in these complex graphs an object in one triple acts as a subject for another. For instance, in the above simple example we can extend the graphs with statements like ‘School has name xyz’ or ‘School has teachers’.

**RDF data and query example**

Let the sample RDF data be –

Below is a more complex RDF graph that has multiple subjects and objects inter-connected via different predicates-

<Book> <title> <Lean Ruby> .

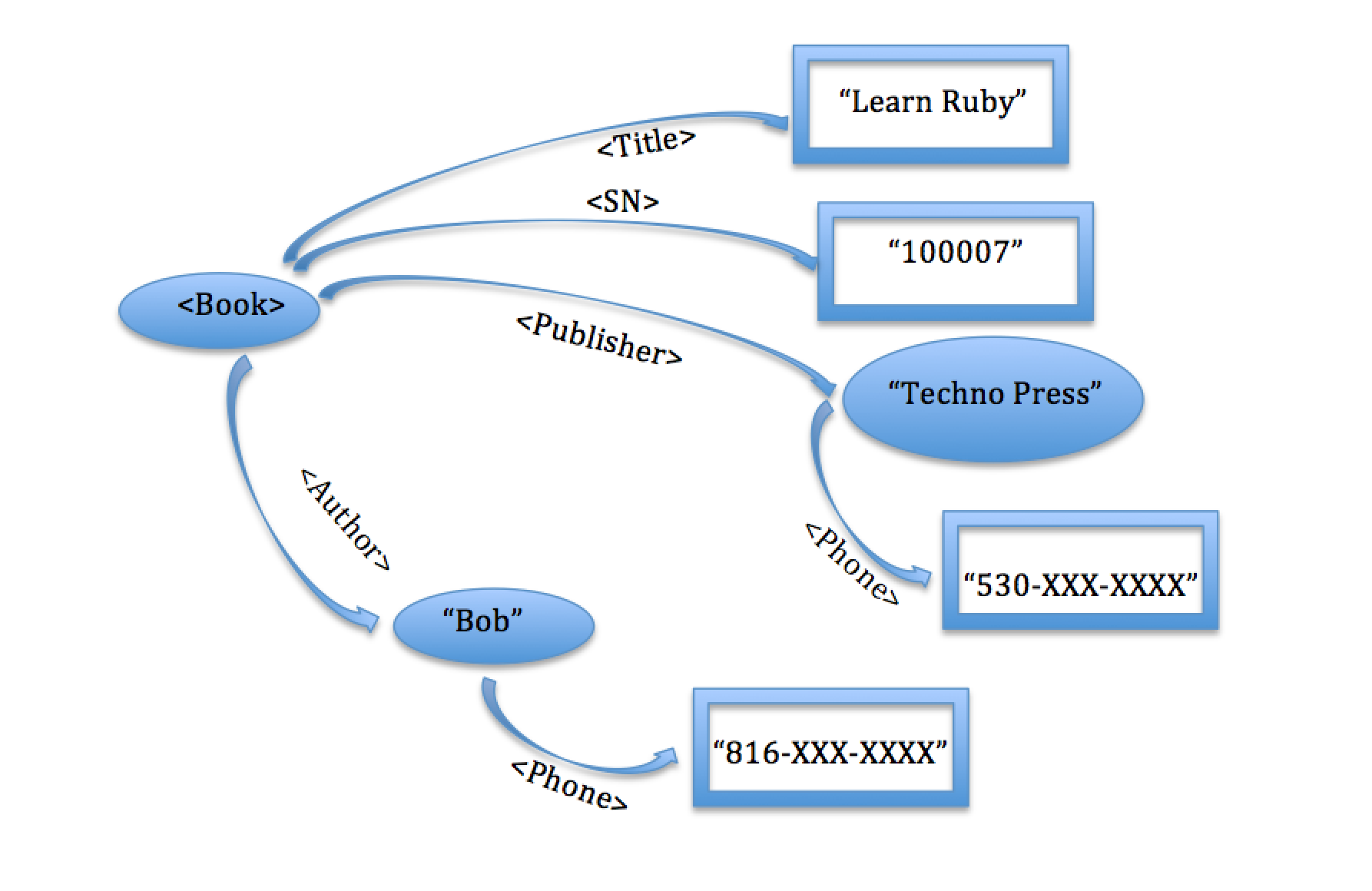
<Book> <sn> <100007> .

<Book> <Publisher> <Techno Press> .

<Techno Press> <Phone> <530-XXX-XXXX> .

<Book> <Author> <Bob> .

<Bob> <Phone> <816-XXX-XXXX> .



**Sample Query**

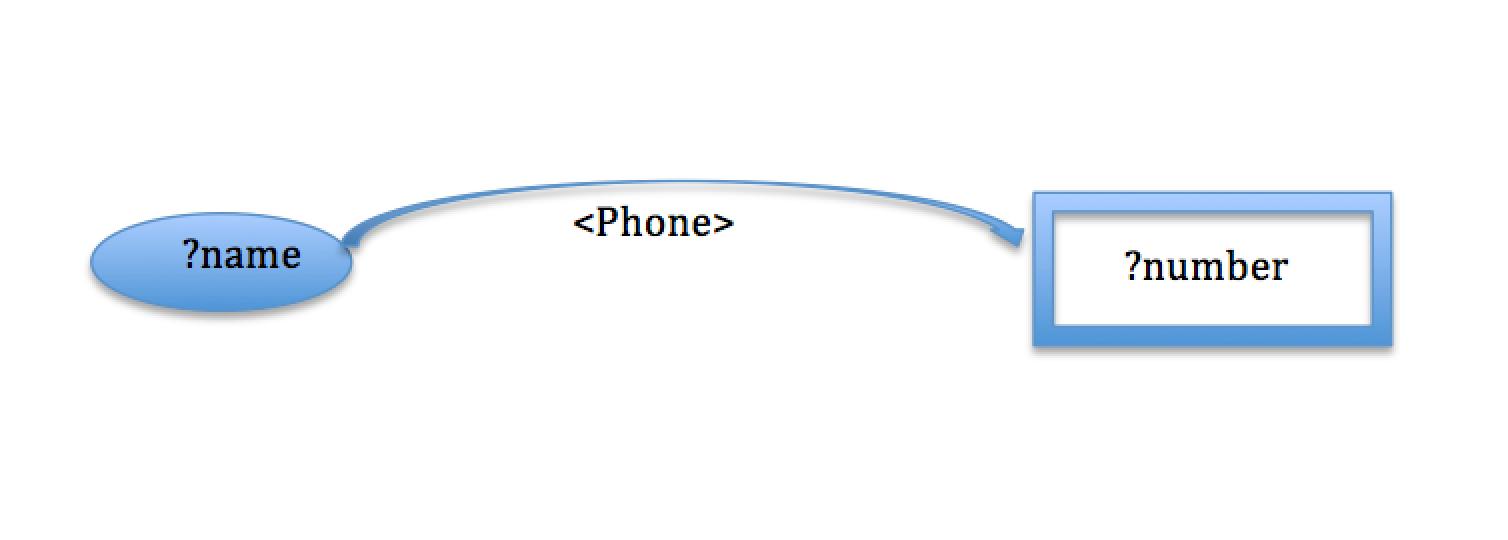
Below is a sample query for the above data. The query has just one triple and two variables. The triple is intended to find all the phone numbers from the data and prints the name and number records accordingly.

SELECT ?name, ?number WHERE

{

?name <phone> ?number

}

****

**Sample Result**

The above query for the given data would fetch two results –

<Techno Press> <530-XXX-XXXX>

<Bob> <816-XXX-XXXX>

The queries for RDF data can be more complex than this, specifically when they have a lot of joins involved. Most of the contemporary querying tools consume a lot of time to return results when the queries are large and have as many as 20 triples in the query. The time consumption to return results also depends on the number of variables involved and the number of self joins with the query. Usually the more number of self joins and variable, much longer the processing will take to return the results.

**Chapter ? Bloom Counter Generation and Comparison**

Bloom Filter

A bloom filter is a memory efficient data structure to determine the probability of the presence of an element in a sample set. It can be used to test whether an element E is a member of a set S or not.

A bloom filter is implemented as a bit array of n bits where the element E's presence in the set is estimated by evaluating if the kth bit where (k < n) is set in the bloom filter or not.

There are two steps involved in the usage of bloom filters –

1) Bloom Filter Generation

In this step, a bloom filter is constructed and its bits are set based on a few operations on the set S. There are many parameters involved in the construction of bloom filter of size N. One important parameter is forming K hash functions H1, H2, H3, . . . HK such that each of those hash functions return a value between 1 and N when applied on an element E.

Initially when a bloom filter with size N is constructed, all the bits in the bit array are set to zero. Then, the value R where R<N which is returned by applying hash function H on an element E represents the bit position that needs to be set in the bloom filter.

Below is an example of how bloom filter B can be constructed –

Let the set S to be inserted be - {3, 14, 19}

Let the size N of the bloom filter B be - 16

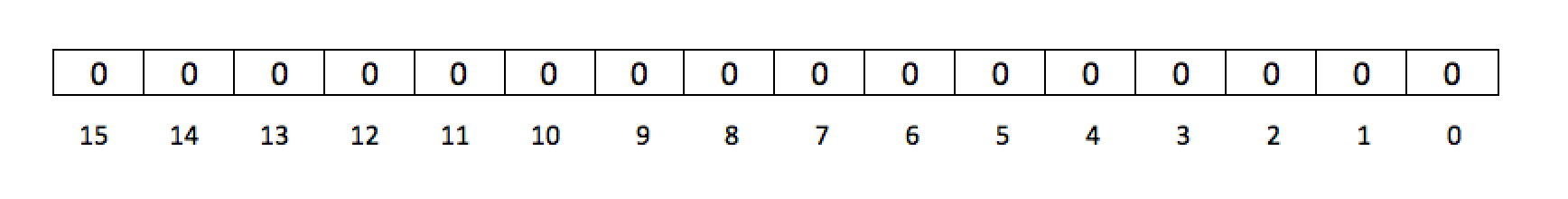
Let the number of hash functions be - 3

Let the hash function H1 be - ((5x + 4) % 20)%16

Let the hash function H2 be - ((9x + 10) % 20)%16

Let the hash function H3 be - ((2x + 11) % 20)%16

1. Initial values in a bloom filter B with size 16

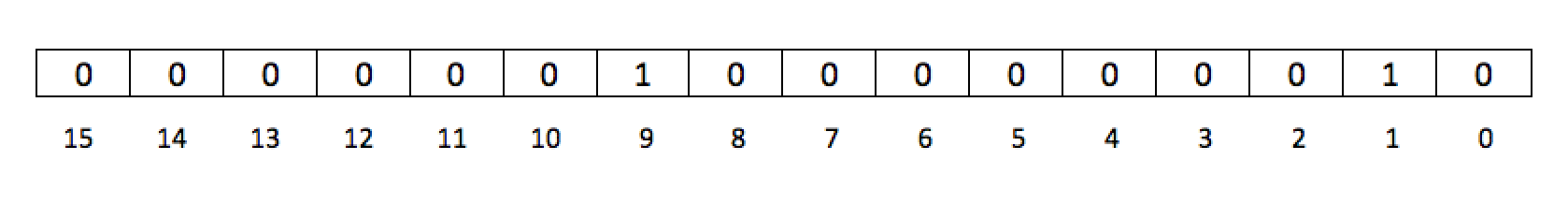


2) Inserting the results of hash functions set on the first element 3

R1(3) = ((5\*3 + 14) % 20)%16 = 9

R2(3) = ((9\*3 + 10) % 20)%16 = 1

R2(3) = ((2\*3 + 11) % 20)%16 = 1

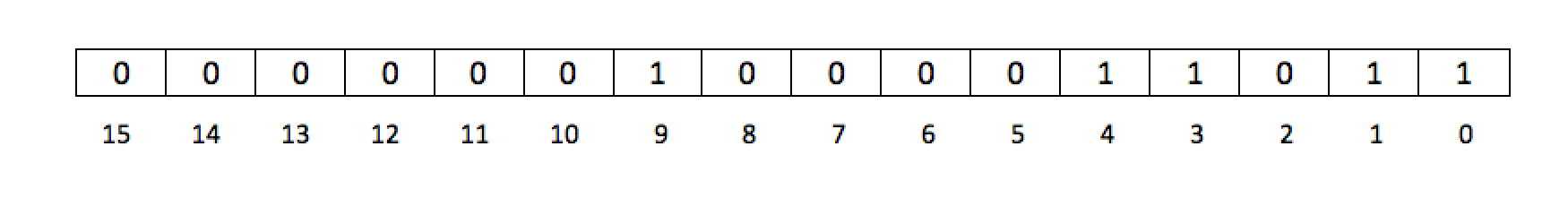


3) Inserting the results of hash functions set on the second element 14

R1(14) = ((5\*14 + 14) % 20)%16 = 4

R2(14) = ((9\*14 + 10) % 20)%16 = 0

R2(14) = ((2\*14 + 11) % 20)%16 = 3

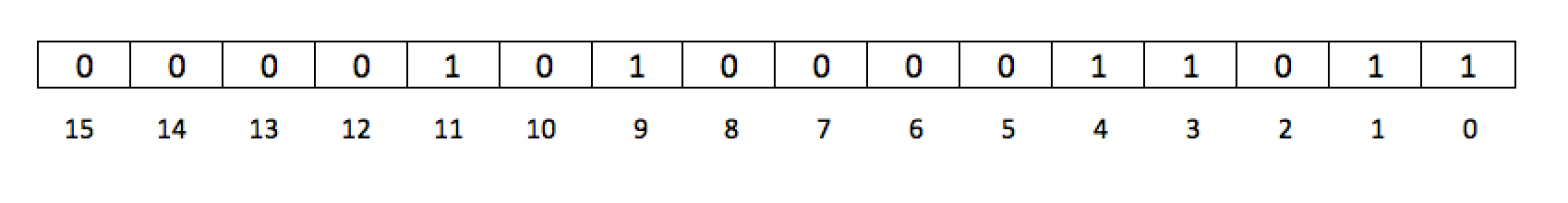


4) Inserting the results of hash functions set on the third element 19

R1(19) = ((5\*19 + 14) % 20)%16 = 3

R2(19) = ((9\*19 + 10) % 20)%16 = 11

R2(19) = ((2\*19 + 11) % 20)%16 = 9



2) Bloom Filter Comparison

Once the bloom filter for set S is generated, bloom filter comparison operation is used to find out the possibility of a query set S2 to be a subset of set S. In this step, another bloom filter B2 is constructed out of the query set S2 using the same set of hash functions H1, H2, H3, . . . Hk used in the first operation.

After we construct the query bloom filter Q, we compare it with the data bloom filter B to check if the bits set in bloom filter Q are also set in bloom filter B. If they are set in B as well, then we say that we have a potential candidate match. Otherwise, if the bits set in Q are not set in B, we can safely assert that the query set S2 is not a part of the data set S.

Below is an example of how query set S2 is computed and compared to determine the possibility of it being a part of data set S -

Let the query set S2 be - {3, 11}

The size N of the bloom filter B is - 16

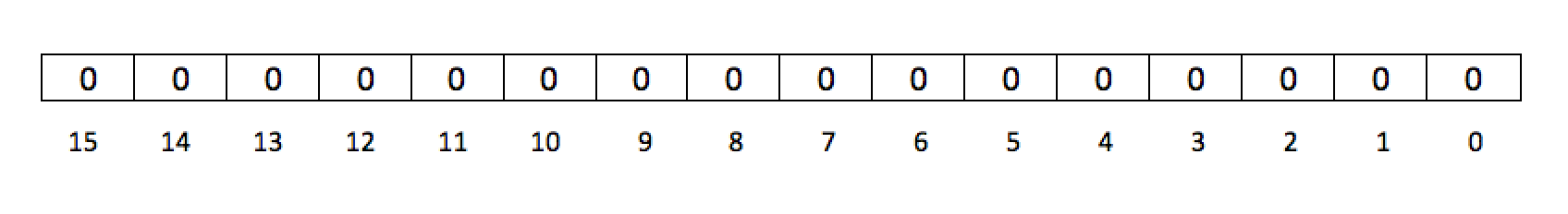
The number of hash functions is - 3

The hash function H1 is - ((5x + 4) % 20)%16

The hash function H2 is - ((9x + 10) % 20)%16

The hash function H3 is - ((2x + 11) % 20)%16

1. Initial values in a bloom filter B with size 16

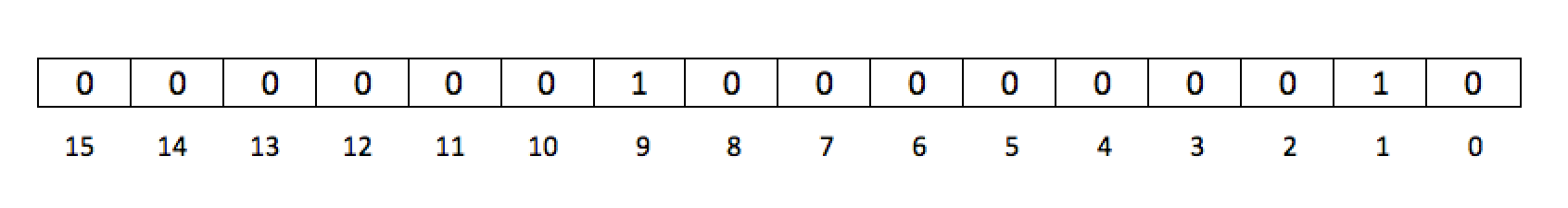


2) Inserting the results of hash functions set on the first element 3

R1(3) = ((5\*3 + 14) % 20)%16 = 9

R2(3) = ((9\*3 + 10) % 20)%16 = 1

R2(3) = ((2\*3 + 11) % 20)%16 = 1

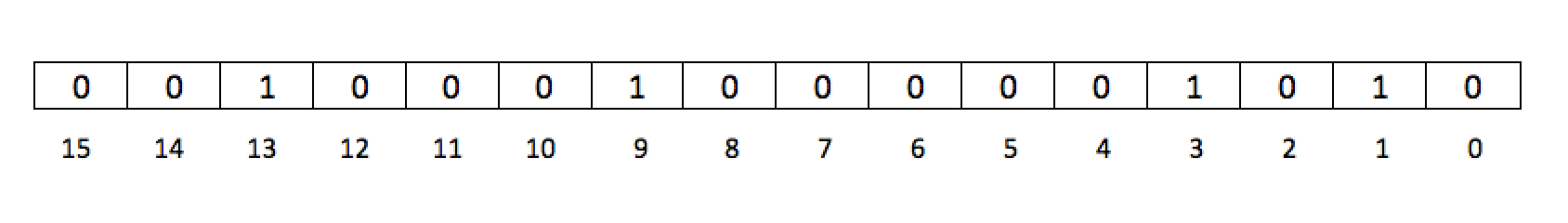


3) Inserting the results of hash functions set on the second element 11

R1(11) = ((5\*11 + 14) % 20)%16 = 3

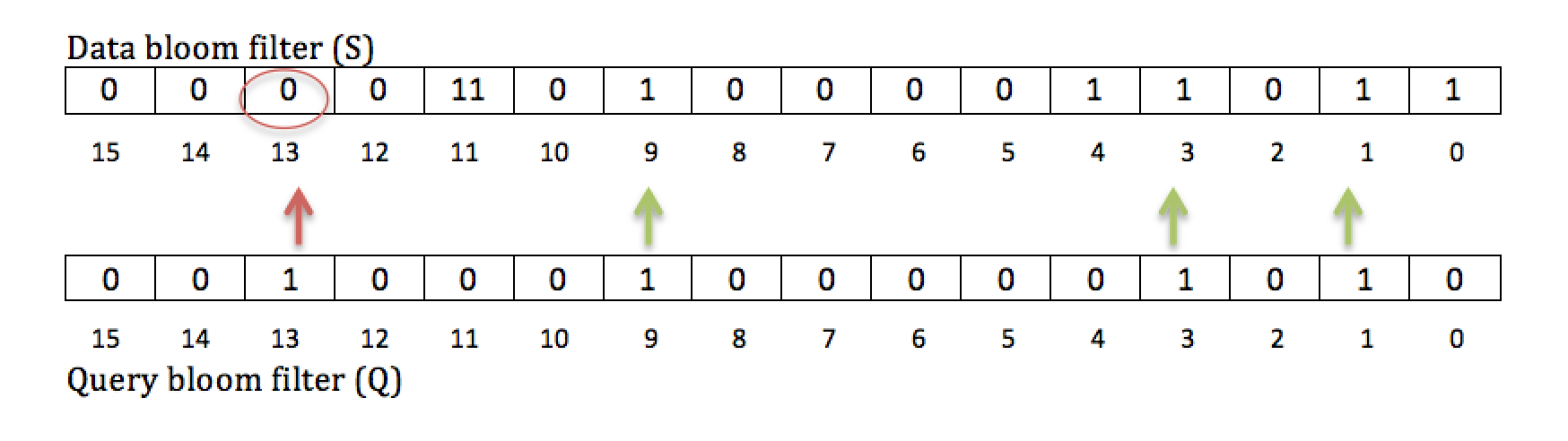
R2(11) = ((9\*11 + 10) % 20)%16 = 3

R2(11) = ((2\*11 + 11) % 20)%16 = 13



4) Comparing the query bloom filter Q with data bloom filter S

The comparison of the query bloom filter Q with data bloom filter S is show as below. For each bit from 0 to 15, the set bits in Q are being checked with S. If that bit is also set in S then the algorithm moves to the next set bit in Q otherwise, it is confirmed that the query returns false.



In the above example, the values for set bit locations in 1, 3 & 9 were matched in both the query set Q and data set S. But there was a difference in the bit located 13 where it was set for query bloom filter Q but not in the data bloom filter S. Hence, it is confirmed that the query set S2 is not part of the data set S, which is true.

Limitations of Bloom Filters and Multi-sets

One of the main drawbacks using bloom filters is it can sometimes be impossible to distinguish between two different multi-sets using the same set of elements. In such cases, bloom filters return guaranteed false positives. A false positive is a result where bloom filters return a true value for a query but, when the query is run on the actual dataset, it returns no results. One common scenario where bloom filters return a false positive is when multi-set query is used on a multi-set data where the set elements of query is a subset of set elements of the multi-set data.

For example, below is a multi-set data and multi-set query, which use the same elements – 3,6,8

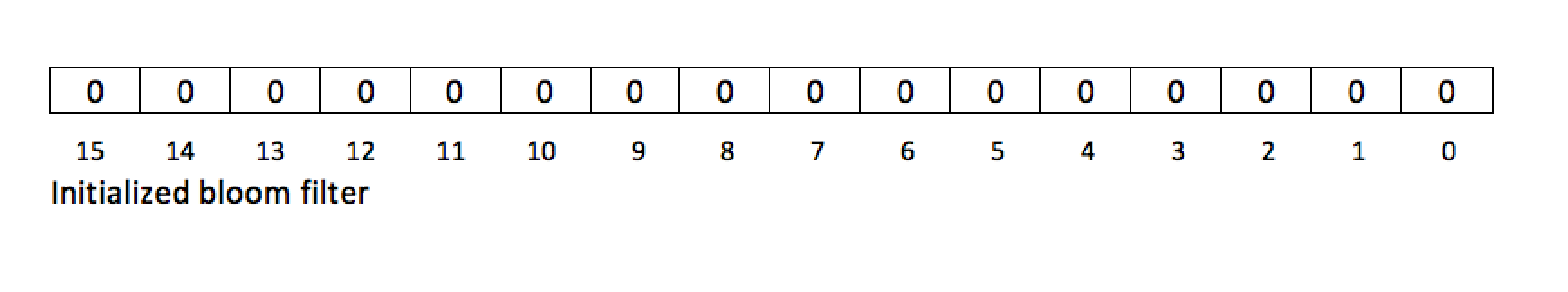
Data Multi-set S = {3, 3, 6 , 8, 8}

Query Multi-set Q = {3, 6, 6, 6}

In the above example, clearly the query multi-set Q is not a subset of data multi-set S because Q has three instances of element-8 where as S has only two. Let us try to apply the above sets on a simple bloom filter -

**Bloom Filter Construction**

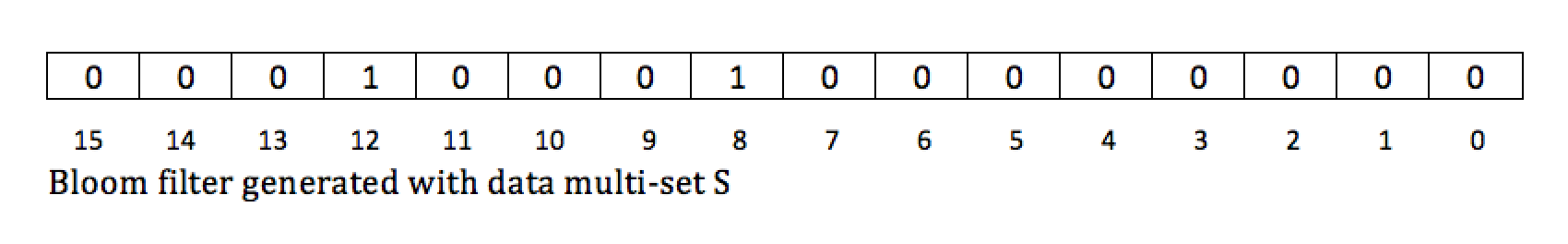
Let the size N of the bloom filter B be - 16



**Bloom Filter for Data Multi-set S {3, 3, 6, 8, 8}**

Let the number of hash functions be - 1

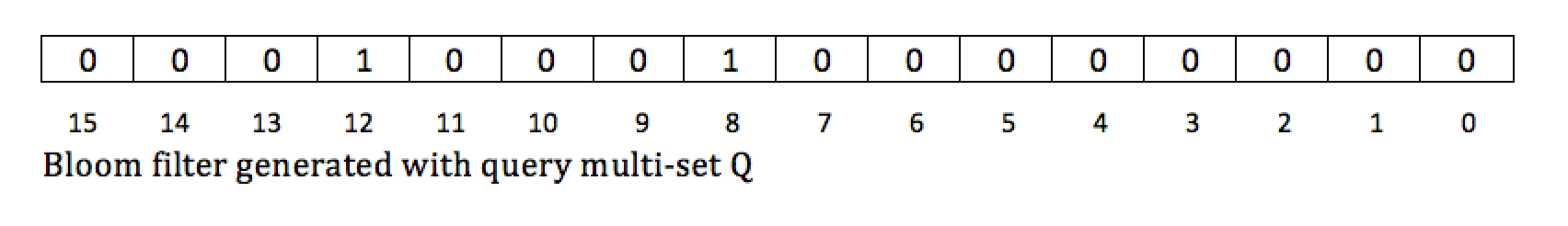
Let the hash function H1 be - ((8x + 4) % 20)%16



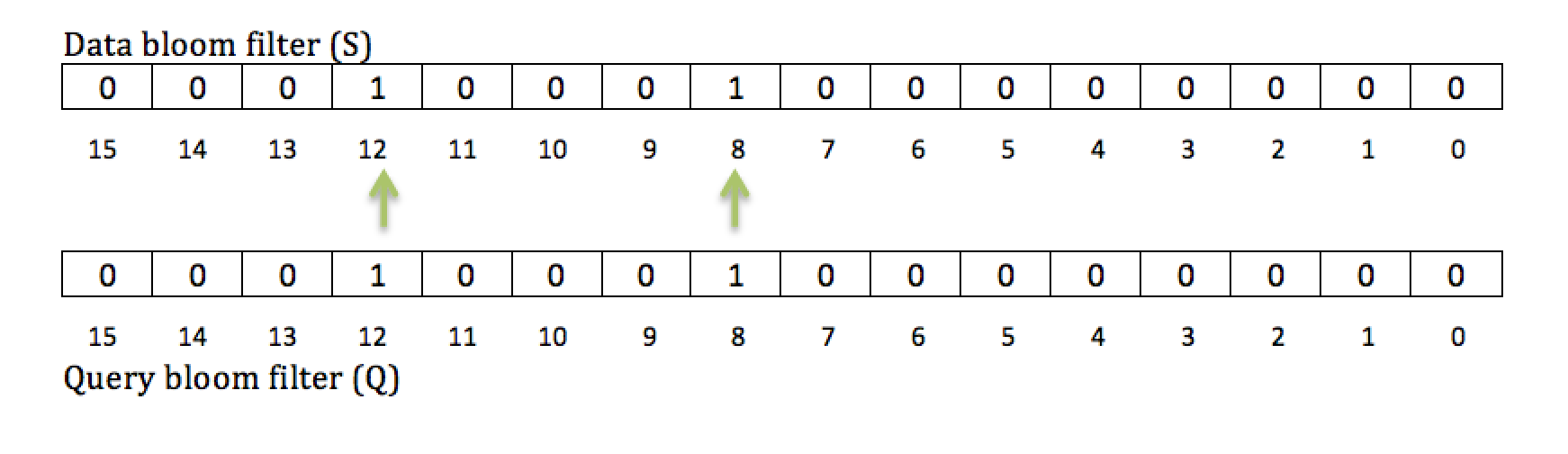
**Bloom Filter for Query Multi-set Q {3, 6, 6, 6}**

Let the number of hash functions be - 1

Let the hash function H1 be - ((8x + 4) % 20)%16



**Comparing Query bloom filters for Multi-set Q with Data Multi-set S**

****

The comparison returns true that Q is a subset of S while it is not. Therefore, this results in the potential candidate match to be a false positive. Hence, it is shown that bloom filters fail to operate affectively on multi-sets.

Counting Bloom Filters

A counting bloom filter is an extension of the regular bloom filter. It is also an array where bits can be set to determine the probability of existence of elements in a set. One of the major differences between counting bloom filters and bloom filters is counting bloom filters dedicate multiple bits in the array for each return value from the hash set where as bloom filters dedicate only one bit in the array for each return value from the hash set. Dedicating multiple bits in the array for each return value from the hash set enables counting bloom filters to count the number of times that value was returned for the whole data set until the counter is full. The values returned after the counter is full will overflow and can be ignored in the implementation.

The ability of counting the number of times the hash set has returned a value helps counting bloom filters to be more affective than the bloom filters and makes them functionally useful for even for multi-sets. For counting bloom filters to be affective against multi-sets the number of bits dedicated for each return value should be in proportion to the average number of repeated instances of an element in the multi-set.

Counter Comparison

During the comparison of query counting bloom filter with data counting bloom filter, the query multi-set is considered a subset of data multi-set as long as the counter value in the query filter is less than or equal to that of the data filter for each counter position. Then, the query would return true and the data multi-set is considered as a potential candidate for results.

If the counter value in the query filter is more than that of the data filter for at least one counter position, then it is asserted that the query multi-set cannot be a subset of data multi-set and the query returns false, thus confirming zero results.

Now, we show how counting bloom filters can overcome the limitations of bloom filters by using the above example where using a bloom filter resulted in a false positive.

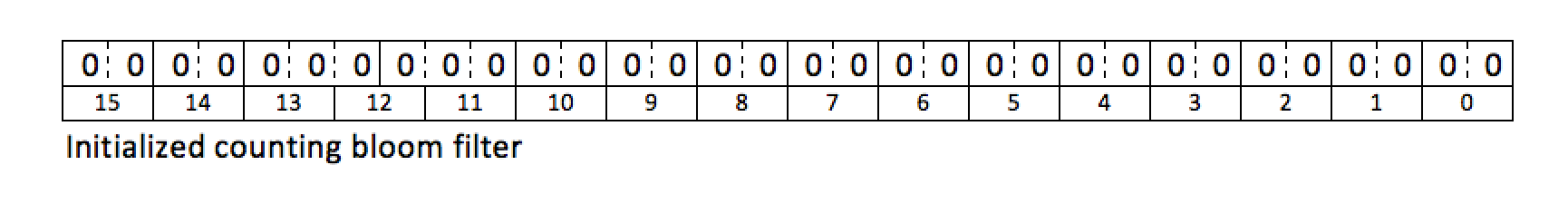
Data Multi-set S = {3, 3, 6, 8, 8}

Query Multi-set Q = {3, 6, 6, 6}

**Counting Bloom Filter Construction**

Let the size N of the bloom filter B be – 16

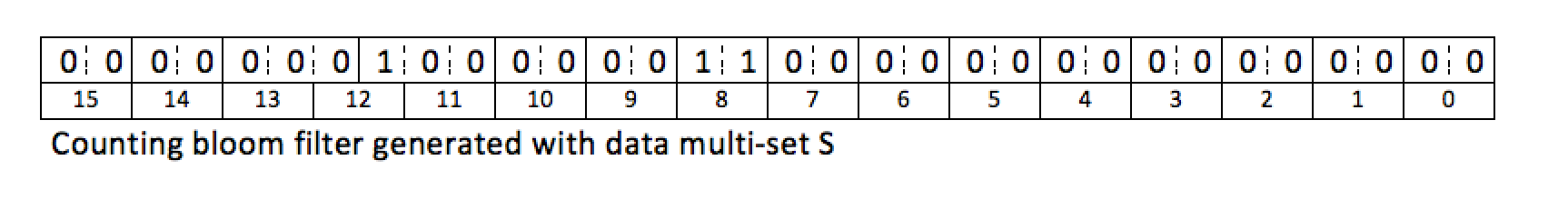
Let the counter size be 2 bits



**Counting Bloom Filter for Data Multi-set Q {3, 3, 6, 8, 8}**

Let the number of hash functions be - 1

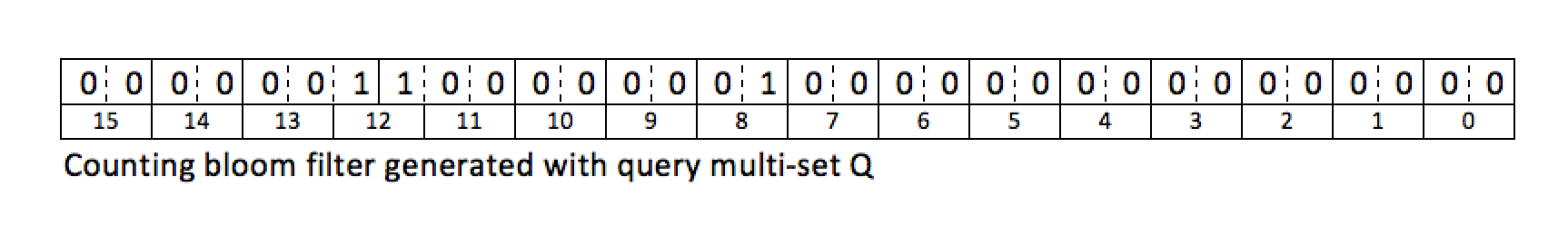
Let the hash function H1 be - ((8x + 4) % 20)%16



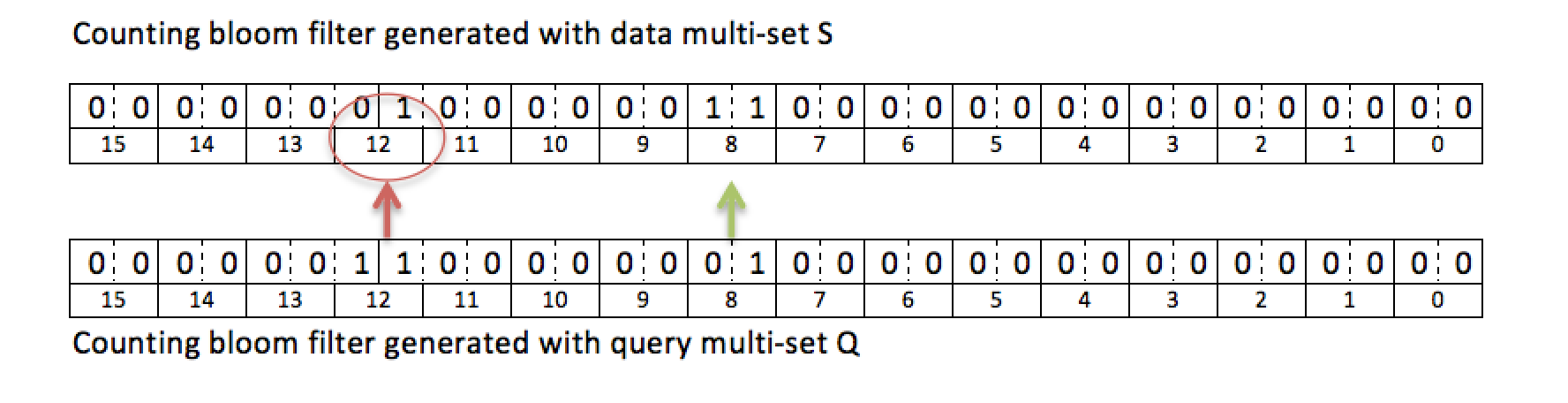
**Counting Bloom Filter for Query Multi-set Q {3, 6, 6, 6}**

Let the number of hash functions be - 1

Let the hash function H1 be - ((8x + 4) % 20)%16



**Comparing counting bloom filters for Query Multi-set Q with Data Multi-set S**



In the above figure, the value in the 12th position in query filter is more than that of in data filter. It can be asserted that the query multi-set is not a subset of the data multi-set, therefore the query returns zero results.

**Bloom Parameters and false positives**

**Chapter? System Architecture**

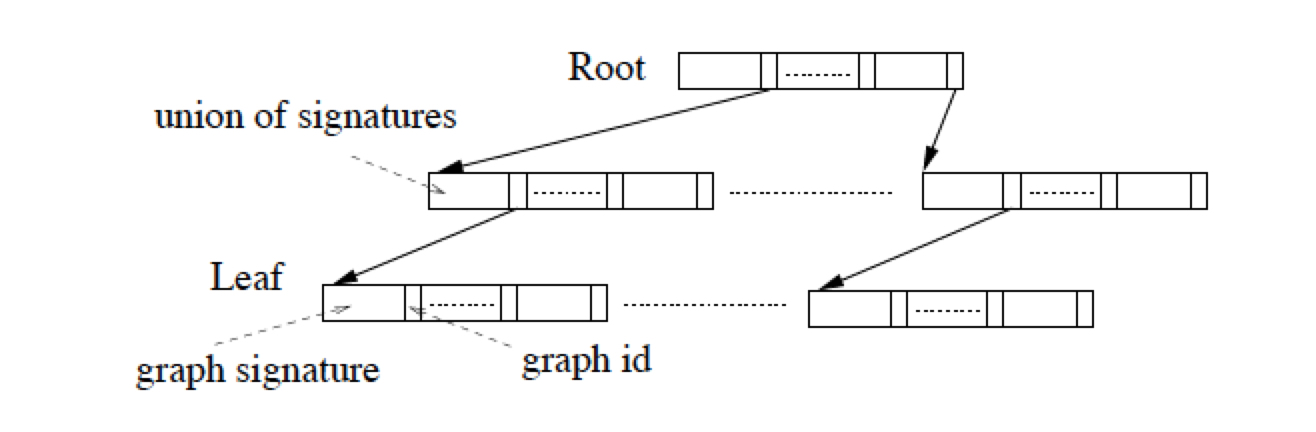
**Chapter? Index Generation**

* RIS tree Generation
* Application of counting bloom filters

**Signature Trees**

1. Storing graph signatures in the Berkley DB

Tree Details:

1. Just like R-trees, our tree also groups similar signatures together in a hierarchy fashion.
2. The tree groups “similar signatures” in the each hierarchy.
3. Node structure is – < sig, ptr> where sig is signature for leaf nodes or union of child signatures for non leaf node and ptr is a pointer to graph id for leaf nodes or a pointer to a child node in the case of non-leaf nodes. 
4. Split method in the algorithm is called only to create leaf nodes with the given fan-out size.
5. Fan-out is defined as the maximum number of signatures in a leaf node.

**Chapter? Querying Index**

Querying index with a given query is one of the most important processes involved in our system. One of the main goals this querying approach is to quickly identify and eliminate those parts of the dataset which we are sure will not return any results. This potentially can save a lot of time, as we do not have to run the query through entire dataset.

One-way to quickly eliminate those graphs which would not return any results is to run the query on only those graphs from huge dataset, which we know have the potential to return results. We call such graphs as the candidate matches.

There are a lot of parameters involved for optimizing the system in order to get the best candidate matches. One parameter is to have the number of candidate matches as low as possible. That means that we have successfully identified and eliminated most of the graphs from the huge collection graphs in the dataset. Another important parameter for effective querying is to have least number of false positives in the candidate matches. It is also possible that the entire candidate matches which were identified to return results may not be guarantee to return any results. There could be a few graphs which even though are identified as a candidate match could return no results at all when the query is run against that graph. We call such graphs as false positives. Lesser the number of false positives, more efficient is the querying approach.

**Candidate Matches**

**Querying candidate matches**

Once the candidate matches are identified, the query is run on each of the graphs sequentially using a standard RDF data-querying tool. We then aggregate the results return by each graph and present it as the out put.

**Chapter? Sample Queries**

We classified RDF queries into two broad categories on the basis of their selectivity. Types of queries

* Low Selectivity
* High Selectivity

Low Selectivity Queries: Those queries, which return high number of results from a large dataset are classified as low selectivity queries. These queries tend to have more variables and less known attributes in them. These queries typically also have less number of joins which make them more generic and thus usually return many results.

Example:

SELECT ?student ?professor

WHERE {

?student <studies\_at> “Unv” .

?professor <teaches> ?student .

}

As the above example shows the query returns all the pairs of students and their professors.

High Selectivity Queries: Those queries, which return low number of results from a large dataset, are classified as high selectivity queries. These queries tend to have lesser number of variables and larger number of uri’s or known attributes in them. Also these queries have many number of joins which make the results much specific and refined.

Example:

SELECT ?student ?professor

WHERE {

?student <studies\_at> “Unv” .

?student <enrolled> “course A” .

?student <batch\_of> “2000” .

?student <teaching\_assistant> “Math101” .

?student <author\_of> “research1” .

?professor <teaches> ?student .

?professor <studied\_at> “unv2” .

?professor <teaches> “Math101” .

?professor <research\_interest> “Web Technologies” .

?professor <author\_of> “research1” .

}

As shown in the above example, the query returns a very specific pairs of students and their professors who are met certain conditions like being the part of the same course, authors of same paper, having same research interest etc.

In the real time scenario, high selectivity queries are more expected as they are more useful while querying a large dataset for a specific information. At the same time, it is more challenging to return results for the query with very high selectivity compared to its counter part. One of the main reasons why running a query with high selectivity is challenging because of the number of joins it contains. Most of the existing query systems use keyword matches to find results and more joins mean more keyword searches. Hence, systems performing big queries take a huge amount of time to return the results compared to smaller ones. Sometimes, the queries can get highly selective and the number of joins can get as high as fifteen to twenty.

Our system has been designed to address specifically this problem of querying large datasets with queries containing many number of joins. One of the important aspect of our system design that helps us overcome this problem is we perform graph based searching rather than purely rely on keyword based searching. In addition to this, our smart graph preprocessing and indexing system gives us a huge advantage with queries which have large number of joins.

We have tested our system a wide range of queries. A few queries contain as many as twenty joins where as few queries contain a about just five joins. We use our system a platform on which a couple of other tools run. We use Jira on the top of our system for queries with high selectivity while we use RDF3x on the top of our system when the queries have low selectivity. We have benchmarked the whole system against a tool called RDF-3X which is considered as a standard to query RDF data.

**Chapter? Experiments**

* Tools used
  + Jena
  + Rdf3x

Questions:

1. What exactly do we store in the Berkley DB? (Union of children signatures for non-leaf nodes?)
2. How do we determine, what signatures to eliminate from querying during the tree traversal for querying?
3. Please explain

“*Rather than computing the similarity over all possible pairs to select the seeds, we only examine the k highest cardinality signatures. The intuition is that if two signatures with high cardinalities share the least number of elements, then the remaining signatures may be more likely to be similar to either one of them*"

1. Why bottom up approach opposed to top bottom in bulk loading?
2. How efficiently can this approach handle updating data?
3. Can I use the algorithm, figures and quotes from the paper?
4. What is seed pick size?
5. How do we traverse graphs for querying and at what stage do we use bloom counters for generating candidate matches?