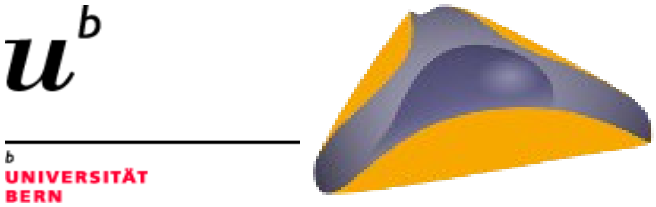

A Comparison of Different Data Structures to Store RDF Data

Master Thesis
by
Rashmi Bakshi

March 2013

Thesis supervisors:
Prof. Philippe Cudre-Mauroux
Marcin Wylot

eXascale Infolab

Department of Informatics
University of Fribourg (Switzerland)

Home University: University of Bern, Faculty of Natural Sciences

Acknowledgements

I have to thank a lot of people in my journey of academics. It would not have been possible to complete my master thesis without the support from my supervisor Prof. Dr. Philippe Cudre-Mauroux and Research Assistant Marcin Wylot.
Philippe was very positive and encouraging about my intention of doing thesis under his supervision. That is the kind of positivity I exactly needed to begin with.
Marcin was my backbone in the whole process.
Regular weekly meetings with him were very helpful as it cleared lots of my doubts and helped me to understand the concepts better and also fixed some technical issues. I really enjoyed working with him because I always learnt something new after leaving his office.
I have to thank my family, my father, who has always been very supportive with his unconditional love and faith in me. There is no way that I could have successfully achieved this milestone without their support emotionally and financially. ☺
Last but not the least my friends/collegues who started their studies with me at the same time. We were diving on the same boat. We all understood each other's condition very well and supported each other at the time of despair.

Abstract

Main focus of my work was to compare data structures based on memory consumed by them during insertion and retrieval of elements(URIs). They were also compared on the basis of CPU time taken to insert and retrieve data. Hash Functions were compared on the basis of number of collisions they produced. i.e. generating same key for different URIs.Their influence on CPU time consumed by the data structure was also considered for comparison.

Data structures considered were:

- **Gnu::hash_map (unordered map)**
- **Google:Sparse_Hash_Map**
- **Std::map(ordered)**
- **Std::unordered_map**
- **Std::Tr1::Boost (unordered_map)**
- **Uthash- Hash Table Library in c**
- **K_hash – Hash Table Library in c**
- **Lexicographic Tree**

Different external hash functions were used to generate keys for each element
They were :

- **Fowler-Noll-Vo Hash (FNV)**
- **Murmur Hash-64 bit**
- **Sbox Hash**
- **One-at-a-Time Hash (Jenkin's Hash)**
- **Bernstein Hash**
- **Combining hash**
- **Paul Hsieh's hash**

Keywords:

DataStructure, Hash functions, Rdf data, Sematic web, Hash table,Memory usage, Collisions, Hash map, CPU cycles, CPU Time,Memory,Speed,Performance

Table of Contents

List of Figures

List of Tables

1 Introduction

As Internet and World Wide Web grew in size, need for digitised data storage also increased. This digitised data need to be stored in an efficient way so it can be retrieved and provided to masses as fast as possible. It has to be identified and categorised.
Just like our traditional library system where books are categorised according to their subject, author, year and book number.
And thus we are going to talk about "Data Structures" that stores the digitised data in a logical manner and provide mechanisms to retrieve it.
Digitized data can be used to generalise electronic data but we need to know precisely what kind of data are we going to store.

1.1Types of data:

- **Structured Data**
  are those that has a pre defined schema i.e. Data that can be organised in a structure. Such as data in Relational Databases for example Oracle Database Management Systems, Mysql etc.Data is organised in a table like format with columns as attributes and rows as values.
  Positives of relational databases is easy retrieval that is high performance and easy to navigate in between the tables .It is good for transaction systems that need to follow ACID properties like e-commerce websites.
  Negatives of them are inflexibility. It is hard to reform already defined schema of a database. Most of the times it has to be redesigned from scratch in cases where data grows enormously and database specifications need to be altered to accomodate new data.
  It is also not scalable. It is not easy to add extra row value without having a subsequent column for it. There are cases where row value might not need an extra column. And thus it is a waste of memory space. An example could be products on Amazon. It is not necessary that all products under same category will have exact number of attributes. Some might have more attributes and some might have less. So how do we accomodate these differences in products of same type in one single table.
- **Semi Structured Data**
  is also called self describing data. Such as xml format that has its own tags.
  It doesnot require any pre defined schema. It doesnot mean that definition of schema is not possible. It is rather optional. [1]
  Positives are that it is flexible and scalable i.e. it is easy to change schema . For example: An instance of an object can have more than one data type. It can accomodate variations in the structure.
- **Unstructured Data**
  There is no structure to identify and distinguish data. Such as html/ images/ videos / text files.
  It's advantage is that no effort is required in its classification . It is also very flexible and can accomodate additional data easily.
  It's disadvantage is that no contolled navigation is possible.
  It's retrieval is basically based on full text search (apache lucene – text retrieval libraries) that can be applied to all kinds of text documents. [1]

Semantic Web is an effort to make web intelligent by modifying the structure of data on the web so that it can be easily linked, processed and delivered to the users.Most of the data on the web is unstructured or semi structured. Semantic Web is an initiative to convert this data into "web of data" build upon RDF Framework.

1.2 RDF Schema

We stored RDF URIs in different data structures and called it RDF data.
It is a type of semi structured data. It is built upon RDF schema which is based on the Web Ontology Language(OWL) where all RDF classes and properties are stored that describe entities and their relationships.
It is a fairly new standard for data exchange on web . It can be referred as database of web. It uses URIs to link data i.e. to describe relationships between entities.
RDF URIs identify the real objects. They are different from HTTP URI that identifies HTML documents. It is just a way to distinguish data at human end (HTML) from data at machines (RDF) [16]

Figure 1.2.1 Rdf schema

Rdf schema consists of subject-predicate-object expressions which are also called triplets where subject represents the resource(thing), predicate means property or an atribute of the resource and object means the ultimate value. For Example,
Book- Title-"Computer Science"
Book- Author-"XXX"
Book is the subject.Title and Author are respective properties of the subject.Computer science and XXX are ultimate values or objects.
Now lets think in terms of URI.
In URI, different objects can have common subject(domain name), may even have common properties.
Nodes(things/subjects) are connected on the basis of these common properties.

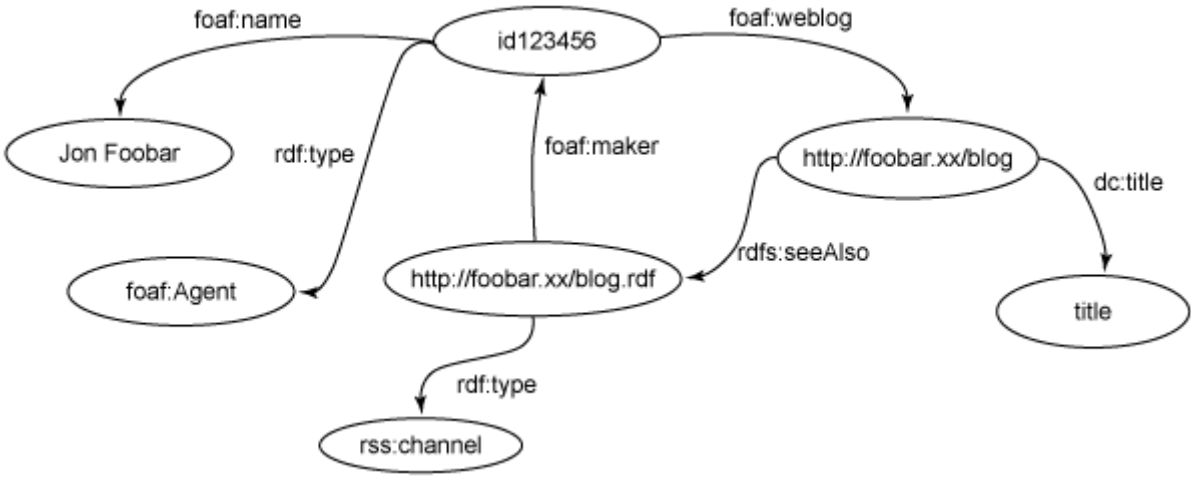

Figure 1.2.2 It is an image from FOAF project which links subjects and objects based on common properties such as foaf:name,foaf:weblog,rdf:type etc

1.3 Motivation and Milestones

Storage, management and retrieval of data over World Wide Web are few of the biggest challenges in the field of computer science today.
WWW is hugely scattered with millions of documents.
The motivation of this thesis is to find out the best available data structures to store URIs of rdf data.
To achieve this, the milestones of the project are

- Develop a basic framework for memory consumed by the process while adding elements to a data structure and retrieving them
- Choose different data structures and add it to the framework
- Implement different external hash functions to the data structures
- Develop a framework to detect collision so as to analyse the quality of hash functions
- Apply data structures to different data sets varying in size
- Compute the results and benchmark different data structures and hash functions

1.4. Notations and Conventions

Formatting conventions:
Bold is used for heading and sub headings,figures and tables. It is also used to emphasise importnat notes in the report.
Italic and Bullets are often used for sub types.
The present report is divided into Sections. Sections are further broken down into sub Sections.
Subsections might contain some Paragraphs.
Figures, Tables and Listings are numbered inside a Section. For example, a reference to Figure1 of Sub Section 2.1 will be noted as Figure 2.1.1.
Source code is displayed as follows:

```
for (i = lines.begin(); i != lines.end(); i++) {
string str = *i;

uint k = Hash(str);

ret = m.insert(pair<uint, string> (k, str));

if (ret.second == false and str != ret.first->second) // pair::second is
set to false if an element with same key existed

{//cout << "element " << k << " already existed";//cout << " with a value
of " << Hash(ret.first->second)<< endl;value++;

}

}
```

2 Memory Consumption Framework

A framework was designed to compute memory consumed by the process while inserting elements to a data structure and while retrieving them back. The idea behind is to compare quality of data structures on the basis of memory consumed by them. Lesser the memory consumption, efficient will be the data structure.

2.1 Proc File System/ VmData

Libproc-dev package was installed in order to use proc file system.
The proc file system is a pseudo file syestem which is used as an interface to kernel data structures in linux.
Most of files in it are read only but few can be changed.
/proc/[pid]:It represents id of each running process.
Ps aux|less command can be used to view all running processes on linux.
command ps -U root -u root -N is used to view all processes except the ones running at root.
pidof <process name> command is used to retrieve the process id of a given process.[3]
For example pidof eclipse
result= 567

```
rashmi@ubuntu:~$ ps -U root -u root -N
  PID TTY          TIME CMD
  458 ?        00:00:00 rsyslogd
  473 ?        00:00:00 dbus-daemon
  497 ?        00:00:00 avahi-daemon
  498 ?        00:00:00 avahi-daemon
  980 ?        00:00:00 colord
 1718 ?        00:00:00 rtkit-daemon
 1924 ?        00:00:00 gnome-keyring-d
 1933 ?        00:00:00 gnome-session
 1966 ?        00:00:00 ssh-agent
 1969 ?        00:00:00 dbus-launch
 1970 ?        00:00:02 dbus-daemon
 1972 ?        00:00:00 gvfsd
 1977 ?        00:00:00 gvfs-fuse-daemo
 1988 ?        00:00:01 gnome-settings-
 1999 ?        00:00:00 gconfd-2
 2001 ?        00:00:00 gsd-printer
 2005 ?        00:00:02 metacity
 2007 ?        00:00:00 gnome-screensav
 2013 ?        00:00:07 pulseaudio
 2016 ?        00:00:00 gconf-helper
 2017 ?        00:00:02 unity-2d-launch
 2018 ?        00:00:00 unity-2d-panel
 2020 ?        00:00:00 dconf-service
 2029 ?        00:00:01 nautilus
 2031 ?        00:00:00 polkit-gnome-au
 2033 ?        00:00:00 bamfdaemon
 2034 ?        00:00:00 gnome-fallback-
 2035 ?        00:00:00 bluetooth-apple
 2041 ?        00:00:00 gvfs-gdu-volume
 2044 ?        00:00:00 nm-applet
 2054 ?        00:00:02 vmtoolsd
 2056 ?        00:00:00 gvfs-gphoto2-vo
```

```
 2062 ?        00:00:00 gvfs-afc-volume
 2068 ?        00:00:00 notify-osd
 2079 ?        00:00:00 dbus
 2092 ?        00:00:00 telepathy-indic
 2096 ?        00:00:00 mission-control
 2112 ?        00:00:00 gvfsd-trash
 2118 ?        00:00:00 unity-panel-ser
 2120 ?        00:00:00 gvfsd-burn
 2132 ?        00:00:00 indicator-sessi
 2134 ?        00:00:00 indicator-datet
 2136 ?        00:00:00 indicator-messa
 2142 ?        00:00:00 indicator-sound
 2143 ?        00:00:00 indicator-appli
 2162 ?        00:00:00 geoclue-master
 2179 ?        00:00:00 gdu-notificatio
 2202 ?        00:00:00 gvfsd-metadata
 2205 ?        00:00:00 zeitgeist-datah
 2211 ?        00:00:00 zeitgeist-daemo
 2212 ?        00:00:00 cat
 2222 ?        00:00:00 applet.py
 2225 ?        00:00:01 unity-2d-places
 2245 ?        00:00:00 unity-applicati
 2246 ?        00:00:00 unity-files-dae
 2248 ?        00:00:00 unity-music-dae
 2284 ?        00:00:00 unity-musicstor
 2365 ?        00:00:00 update-notifier
 2387 ?        00:00:03 update-manager
 2435 ?        00:00:00 deja-dup-monito
 2456 ?        00:00:00 deja-dup
 2570 ?        00:00:00 gnome-terminal
 2577 ?        00:00:00 gnome-pty-helpe
 2578 pts/0    00:00:00 bash
 2634 pts/0    00:00:00 ps
rashmi@ubuntu:~$ pidof colord
980
rashmi@ubuntu:~$ cd /proc
rashmi@ubuntu:/proc$ ls
1     1988  2092  2365  4    882          fb             partitions
10    1999  2096  2383  458  883          filesystems    sched_debug
1006  2     21    2387  473  9            fs             schedstat
11    20    2112  24    493  939          interrupts     scsi
12    2001  2118  241   497  953          iomem          self
1202  2005  2120  2435  498  980          ioports        slabinfo
1203  2007  2132  2456  502  984          irq            softirqs
1230  2013  2134  25    505  989          kallsyms       stat
13    2016  2136  2553  529  992          kcore          swaps
1306  2017  2142  257   588  acpi         key-users      sys
14    2018  2143  2570  589  asound       kmsg           sysrq-trigger
15    2020  2162  2577  6    buddyinfo    kpagecount     sysvipc
16    2029  2179  2578  628  bus          kpageflags     timer_list
```

```
167  2031  22   258  7   cgroups     latency_stats  timer_stats
1679  2033  2202  2775  710  cmdline     loadavg       tty
169   2034  2205  2781  726  consoles    locks         uptime
1718  2035  2211  2850  736  cpuinfo     mdstat        version
19   2041  2212  3    754  crypto      meminfo       version_signature
1924  2044  2222  310   8    devices     misc          vmallocinfo
1933  2046  2225  315   861  device-tree  modules       vmstat
1966  2048  2245  33    865  diskstats   mounts        zoneinfo
1969  2054  2246  35    874  dma         mpt
1970  2056  2248  36    875  dri         mtrr
1972  2062  2284  37    877  driver      net
1977  2068  23    38    880  execdomains  pagetypeinfo
rashmi@ubuntu:/proc$ cd self
rashmi@ubuntu:/proc/self$ ls
attr           cpuset  limits     ns           schedstat      syscall
autogroup      cwd     loginuid   oom_adj      seccomp_filter  task
auxv           environ  maps       oom_score    sessionid      wchan
cgroup         exe     mem        oom_score_adj  smaps
clear_refs     fd      mountinfo  pagemap      stack
cmdline        fdinfo  mounts     personality  stat
comm           io      mountstats  root         statm
coredump_filter  latency  net        sched        status
rashmi@ubuntu:/proc/self$ cat status
Name:  bash
State:  S (sleeping)
Tgid:  2578
Pid:  2578
PPid:  2570
TracerPid:  0
Uid:  1000  1000  1000  1000
Gid:  1000  1000  1000  1000
FDSize:  256
Groups:  4 20 24 46 116 118 124 1000
VmPeak:  7376 kB
VmSize:  7376 kB
VmLck:  0 kB
VmHWM:  3528 kB
VmRSS:  3528 kB
VmData:  1832 kB
VmStk:  136 kB
VmExe:  876 kB
VmLib:  2092 kB
VmPTE:  36 kB
VmSwap:  0 kB
Threads:  1
SigQ:  0/7902
SigPnd:  0000000000000000
ShdPnd:  0000000000000000
SigBlk:  0000000000010000
SigIgn:0000000000384004
```

```
SigCgt:         000000004b813efb
CapInh:         0000000000000000
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         ffffffffffffffff
Cpus_allowed:           ff
Cpus_allowed_list:      0-7
Mems_allowed:           1
Mems_allowed_list:      0
voluntary_ctxt_switches:        137
nonvoluntary_ctxt_switches:     33
rashmi@ubuntu:/proc/self$
```

Status*(highlighted in yellow)* is a file in self(*highlighted in pink*) sub directory present in proc directory.It is used to compute VmData(*highlighted in green*) which gives memory consumed by actual size of the data. i.e. Number of elements in a data structure in our case. VmSize(*highlighted in red*) gives vitual memory size consumed by the process along with other backend processes. Therefore it is greater than VmData. And Hence, VmData is preferred over Vmsize to compute memory usage in order to get precise results.

2.2 Total CPU Time Consumed

getrusage() function built in linux kernel was used that returns resource usage measures for RUSAGE_SELF, RUSAGE_CHILDREN and RUSAGE_THREAD We used it for RUSAGE_SELF returns resource usage statistics for the calling process, which is the sum of resources used by all threads in the process.

struct rusage {

```
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

RUSAGE_CHILDREN as the name suggests gives resource usage stastics for all the children of the calling process that have been waiting or terminated.

RUSAGE_THREAD gives usage statics for a single thread. [4]

2.3 Time Taken to Load Data

The GNU C Library provides two data types for representing an elapsed time. They are used by various GNU C Library functions, and we used one of them to calculate time taken to load data. ie. Time taken to insert elements in a data structure.
data type: struct timeval and data type: struct timespec
They are exactly the same except that one has a resolution in microseconds and the other, newer one, is in nanoseconds.
We used timeval structure to represent an elapse time. It is declared in sys/time.h and contains following data members.
long int tv_sec: It represents the number of whole seconds of elapsed time.
long int tv_usec: It is the rest of the elapsed time (a fraction of a second), presented as the number of microseconds. It is always less than one million.[5]

An instance of timeval struct was passed to gettimeofday() member function in linux that is used to get the current time.

2.4 CPU Cycles

A clock cycle is a single electronic pulse of a processor. Most cpu processes require multiple clock cycles as only basic operations can be performed during each cycle. Thus heavier the task,larger will be CPU cycles. [6]
The frequency of a processor is measured in clock cycles per second. It is also referred as clock speed.
We compute CPU cycles using tsc time stamp counter that counts cycles. However,In order to convert it into time(seconds), we have to divide it by frequency of the cpu. since cpu won't run at a fixed frequency due to power management and since I used virtual machine, where I cannot use inherent intel processor's frequency, we could not convert cycles in time. So we left it as cycles.

time_in_seconds = number_of_clock_cycles / frequency

The RDTSC instruction returns a 64-bit time stamp counter (TSC), which is increased on each clock cycle. It is the most precise counter available on x86 architecture.
The RDTSC instruction allows you to get the cpu's current cycle count but bear in mind that you can never get an exact count of how many cycles some process actually uses, since the operating system and other programs running in the background will consume cycles too.

CPU cycles are important to determine processor's overall performance but it is not the only factor. Since processors have different instruction sets, they might differ in the number of cycles needed to complete each instruction. Therefore, some processors can perform faster than others even at slower clock speeds.

RDTSC did not give reliable number of cpu cycles. It was deviating too much plus it was negative in cases of some data structures such as Sparsehash map, Std_map and Boost .

Hash map with fnv hash function gave results for number of cpu cycles as 18446744071363226730

Since the number could not be relied upon because of its size,we decided to use PAPI and then compare its result with RDTSC and choose the better one. PAPI, in no doubt was much more reliable as it gave consistent number of CPUcycles.

PAPI is an acronym for Performance Application Programming Interface. It provides an API

for accessing hardware performance counters present on most processors.[7]

Header file, helper_papi_tracer.h was used which is one of the component of PAPI-C library.

An instance of PapiTracer class was created which invoked a method called start. The method initialised the current event and started the timer. It returned the integer value of clocks per second.

Elements were added into a data structure as a <key,value> pair where key was computed by passing the value to a hash function.

After this operation, stop method was called by an instance of papitracer class that returned std::pair of number of cpu cycles and cpu instructions.

A snippet of code is shown below.

```
#include "helper_papi_tracer.h"

PapiTracer p;

//read data from a dataset and add to a vector

int events = p.start(); // Default parameter is PAPI_TOT_INS, CPU cycles
are always measured

//traverse through vector,Compute key, detect collision and Insert
<key,Value> in a Data Structure

PapiTracer::result_t result = p.stop(events);

std::cout << "CPU Cycles " << result.first << std::endl;

std::cout << "Total Instructions " << result.second << std::endl;
```

3 Data Structures

Data structures are basically used to store data and retrieve it back when required.
Its basic operations are
Traversing: Accesing each record exactly once so that certain item in the record can be processed.
Searching: Finding the index/location of the record with a given key/id.
Insertion: Adding new record to the structure.
Deletion: Removing a record from a structure. [8]
It can be further classified into two types, Linear and Non Linear
Linear type of data structures include array, stack, queue and linked list where a particular element is retrieved by linear search mechanism where it is compared to each item in the collection.
This approach can be very inefficient when size of the container gets bigger. It will take plenty of time to search for a particular element.

Non Linear:
Graphs and Binary tree are types of non linear data structures which consists of linked nodes, in heirarchial manner in case of a tree.
Binary search is a better approach where one starts searching in middle of the sorted list to check if that value is less than or greater than needed value and then progress either in ascending or descending order.
It is more efficient because it consumes less time as compared to the linear search .
3rd type of data structures are *Hash Tables* that can be called as hybrid of previous two types. *Hash tables* support one of the most efficient types of searching called *Hashing*.
Fundamentally, a hash table consists of an array in which data is accessed via a special index called a key. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a hash function.
A hash function accepts a key and returns its hash coding, or hash value
We basically used hash maps and tables for the storage.
Hash table stores key value pair in an associative array. It is synchronised ,meaning that only one thread can access it at a time,following first in first out(FIFO)order.It is costly and have performance overhead in terms of wasting valuable processing time in case of a single thread. But synchronization is necessary and useful in cases of multiple threads that can interfere and modify the state of the container. Synchronization is implemented mostly for safety reasons. It might or might not be ordered.
Hashmap also stores key value pairs. It is unsynchronised meaning that multiple threads may execute at the same time. If one is sure that container will not be shared between the threads, then hashmaps are good to use. It boosts performance as time is not reserved for synchronization.

3.1 Google::Sparse_Hash_Map (unordered map)

sparse_hash_map<Key, Data, HashFcn, EqualKey, Alloc>
sparse_hash_map is paired associative container that stores key value pair. It is also unique which means that each key in the container is unique.
It also stores element in an unordered manner if order matters, then map is more useful.
Look up is done by its key. it also utilises minimum memory usage but it can be slower than other hash map implementations.
This class is appropriate for applications that need to store large "dictionaries" in memory, and for applications that need these dictionaries to be persistent. [9]

3.2 Sgi/Gnu::hash_map (unordered map)

It stores key value pairs but not in order. It is also unique i.e. two keys cannot be compared equal. Looking up an element in a hash_map by its key is efficient, so hash_map is useful for "dictionaries" where the order of elements is irrelevant. If it is important for the elements to be in a particular order, however, then map is more appropriate. [10]

hash _map is not implemented by standard c++ library. It is represented by sgi extension i.e. gnu library. There is no standard sgi implementation for hash map. It moved to gnu.
sgi site documentation is still running but the code is no longer maintained.
gnu has two namespaces.
<include ext/hash_map>= It uses gnu implementation
<include backward/hash_map>= It uses previous versions

We used gnu implementation for our experiment.

3.3 Std::map(ordered)

Maps are also associative container that store elements formed by the combination of a key value and a mapped value.
Unique key values: no two elements in the map have keys that compare equal to each other. Internally, the elements in the map are sorted from lower to higher key value following a specific strict weak ordering. this is to say that elements are ordered. [11]

3.4 Std::unordered_map

It is similar to std::map just with one difference that is elements are not stored in an order.[12]

3.5 Std::Tr1::Boost<unordered_map>

Previous c++ library data structure such as std::map uses binary trees for implementation so that lookup time has more logarithmic complexity. There are cases where hash tables perform better with constant complexity and thus C++ Standard Library Technical Report introduced the unordered associative containers which are implemented using hash tables. Containers are <unordered_set> and <unordered_map>
To store an object in an unordered associative container requires both an key equality function and a hash function
We used <unordered_map> for our experiment. [13]

3.6 uthash- hash table library in c

It is a single header file. uthash.h
In uthash, a hash table is comprised of structures. Each structure represents a key-value pair. One or more of the structure fields builds the key. The structure pointer itself is the value. Note that, in uthash, your structure will never be moved or copied into another location when you add it into a hash table. This means that you can keep other data structures that safely point to your structure regardless of whether you add or delete it from a hash table during your program's lifetime.
There are no restrictions on the data type or name of the key field. The key can also comprise multiple fields, having any names and data types. [14]

3.7 k_hash – hash table library in c

It is a generic hash library in c. It is fast and light weight api.
Keys and Values are kept in separate arrays. It avoids waste of memory when key and value types are different and cannot be aligned together. for example key with int data type and value with string of characters.
Though this strategy can cause consistency problem and misses as keys and values are retrieved twice plus speed can become slow. This is the trade off that has to be considered between speed and memory usage in case of K_hash [15]

4 Hash Functions

Bret Mulvey listed three primary uses for hash functions:

1.Fast table lookup

Fast table lookup can be implemented using a hash function and a hash table. Elements are found in the hash table by calculating the hash of the element's key and using the hash value as the index into the table. This is definately faster than other methods, like examining each element of the table sequentially to find a match.

2.Message digests

It means to compare hash values to determine if they are equal. It is used in cases where data is large size vector and time consuming to compare bit by bit. Instead its hash values are compared. If hash values are the same, then original vectors are likely to be same provided hash function is good and powerful.

3.Encryption

Hash functions can be used to transform data into unreadable format that could be decrypted using secret key. These hash functions are also called cryptographic hash functions. They are usually slow but results in less number of collision. [18]

4.1 Properties of a good quality hash function

1. All good hash functions should avoid collision meaning it should always produce unique keys/hash values for different elements.

2.A good hash function should pass avalanche test. A hash function is said to achieve avalanche if the resulting hash value is widely different even if a single bit is different in the key. This test increases the chances of having unique hash values and minimum collision.

3.A good hash function uniformly distributes its hash values meaning it fills the array/container more evenly. This helps in having less collision as well. [18]

Peter Kankowski classified hash functions required for hash tables into two categories . They are different from cryptographic hash functions because they should be much faster and need not resist intruder's attack.
There are two classes of the functions used in hash tables:

- multiplicative hash functions, which are simple and fast but have a high number of collisions.
- more complex functions, which have better quality but takes more time to calculate.

Multiplicative hash function algorithm

```
"UINT HashMultiplicative(const CHAR *key, SIZE_T len) {
        UINT hash = INITIAL_VALUE;

        for(UINT i = 0; i < len; ++i)

           hash = M * hash + key[i];

        return hash % TABLE_SIZE;

    }"
```

A multiplicative function works by adding together the letters weighted by powers of multiplier. For example, the hash for the word *TONE* will be:
INITIAL_VALUE * M^4 + 'T' * M^3 + 'O' * M^2 + 'N' * M + 'E' “ [19]

I started with multiplicative hash function that used INITIAL_VALUE of 0 and M of 37. It was a poor quality hash function and produced lots of collisions. Source code is given below

```
int hash1(const string &key, int tableSize) {

	int hashVal = 0;

	for (int i = 0; i < key.length(); i++){

	hashVal = 37 * hashVal + key[i];

	hashVal %= tableSize;

	if (hashVal < 0)

	hashVal += tableSize;

	return hashVal;

	}

}
```

I changed it with another multiplicative hash function **Bernstein hash** with initial value of 0 and m=33.

```
unsigned int Hash(string s) {

uint hash = 0;

char c[2000];

for (uint i = 0; i < s.size(); i++) {

c[i] = s[i];

hash = hash * 33 + (uint) c[i];

	}

return hash;

}
```

It worked well and produced less collisions as compared to the first one.I think the reason was table Size as an argument in source code of first hash function.Hash table will have a static size for a given dataset and if it is smaller than the element size then the chances of generating same key for different elements are higher because of “hashvalue%=tablesize”.

Peter mentioned some heuritics for multiplicative hashes. They are

- the multiplier should be large enough to accommodate most of the possible letters (e.g., 3 or 5 is too small);
- the multiplier should be fast to calculate with shifts and additions and therefore it is better if it is an odd number.
- prime numbers are good multipliers. [19]

4.2 Complex hash functions

These functions do a good job of mixing together the bits of the source word thus they are capable of producing completely random results even though elements are similar.

4.2.1 Fowler/Noll/Vo (FNV)

is very powerful hash function with carefully chosen constants following the basic algorithm

```
"hash = offset_basis
for each octet_of_data to be hashed
 hash = hash * FNV_prime
 hash = hash xor octet_of_data
return hash"
```

[20]

4.2.2 Murmurhash-64 bit version 2

It was developed by Austin Appleby who currently works for Google on developing hash functions for cross platform systems. I used murmur hash for 64 bit processor. It utilises an algorithm where bits of values are mixed thoroughly. It uses multiply+shift+xor algorithm. Documentation says that constants such as m and r were carefully chosen by repetitive experiments to be sure that hash value is unique such that it passes avalanche test. It passed "Bob Jenkin's frog.c torture-test." Documentation says that no collisions are possible for 4-byte keys. [21]

We will talk about quality of this powerful hash function based on our experiment in later section of the report.

There is version 3 but it is in the beta form and complete source code is not provided. Probably they are still making changes to it.

4.2.3 Sbox

Sbox means use of substitution boxes in hash funcions. It provides a simple way to confuse the relationship between the input keys and the hash result. It fulfills avalanche criteria. However, note that passing avalanche test does not necessarily means that there will be no collision. This function can consume more memory than the other functions but its implementation is very simple. Theoretically perfect mixing function can be constructed using s box with 2 to the power 32 values but that would consume enormous amount of

memory(128 gb) and thus just 256 values are used by this function where shifting and adding is done to make the hash dependent on the positions of the key bytes.[22]

4.2.4 One-at-a-Time hash

Bob Jenkin designed one at a time hash that reaches avalanche and performs very well. It is recommended to be first few of the hash functions that should be tested.It has been used heavily in scripting languages. [23]

4.2.5 Paul Hsieh's hash

It uses one at a time as a model and Paul claims that it performs 66 % better than Bob Jenkin's One at a Time hash. [24]

We will look if it is true further in our experiment section.

4.2.6 Combining/mixing hash

Here string is divided into 32 bit blocks. Blocks are further broken into partial blocks if necessary. For each block, the combining function is applied to the prior state and the bits from the current block, and then the mixing function is called. This step is repeated for each full-sized block in the string.Then the final, partial block is processed, if necessary. The combining step is modified to accomodate the incomplete block.Finally, some final processing is done to further randomize the internal state. In this case, the mixing step is applied two more times.[25]

Performance of Combining hash was very poor as it produced 82.2 % of collisions in dataset containing 26 million URIs. It is, on an average 50% worse as compared to rest of the hash functions and thus we excluded it from rest of the experiment for fair comparison among almost equally strong hash functions.

5 Related Work

Nick Welch designed benchmark for c,c++ hash tables including Google Sparse Hash Map, Dense map, Std::unorderedmap, Boost, Python's dict, Glib ghashtable,ruby's hash. In common with my experment were Google Sparse Hash Map, Std Unordered Map and Boost. He used a dataset of 40 million with string data type. He used keys as integer and strings. His benchmark comprised of sequential inserts (time taken to insert continous series of integer keys), random inserts (series of random integer keys) and deletion benchmark(time taken to delete key-value pair) [27]
His summary over Sparse hash map, Boost unordered map and std/gcc::unordered_map is as follows.
Sparse Hash Map: It is 2-3 times slower as compared to rest of the data structures but it consumes half of the memory.
Boost Unordered Map: It is second most memory conservative data structure. It performs well and consumes less memory usage upto 10-20 million entries. Beyond that, it's performance decreases significantly.

Gcc::Unordered map: It is stable and will do fine if there are no specific needs and size of dataset is not incredibly large. However, author gave it “the most boring” award being a “compiler provided option”.[27]
Please refer to his article for more details about other hash tables and to view graphs about memory usage by each data structure and CPU time in seconds for each kind of insert. [27]
We will further compare his observations with our results in Analysis section of the report.

5.1 Lexicographic Tree

Marcin's LexicographicTree was also tested to measure it's memory consumption(in kb),cpu cycles and cpu time(in seconds). [31]
It is a tree like structure where common URI(prefix) are the root of the tree and its subsequent suffix are its children.

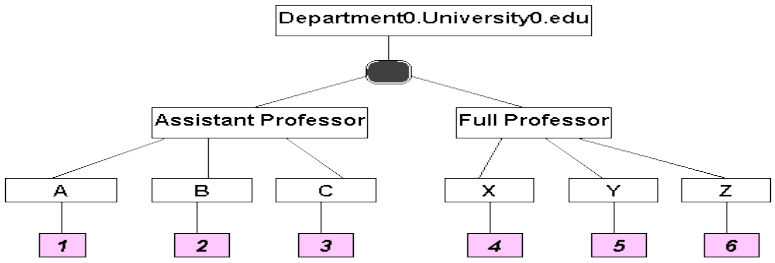

Figure 5.1.1: Diagramatic Representation of Lexicographic Tree

In our dataset we have URIs such as
AssistantProfessorB@department0.university0.edu
AssistantProfessorC@department0.university0.edu
FullProfessorX@department0.university0.edu
AssistantProfessorA@department0.university0.edu
FullProfessorZ@department0.university0.edu

where A, B, X , Y are variables< apparently names of the professors>.
Basic idea behind lexicographic tree is to break the URI into common parts such as domain name(subject) and distinguished parts. It is a hierarchial strucutre that follows top down methodology.(It expands from top to bottom).
Lets take one URI,
AssistantProfessorA@department0.university0.edu
In given above URI, department0.university0.edu is the subject and assistant Professor A is the object. It follows RDF schema that has been discussed in Introduction section of the report.
Each distinguished part of an object such as A,B,C is assigned an identifier which is auto incremented. It is called Leaf.
It can be said that there are no collisions in lexicographic tree because key is auto incremented and always increasing.

Note: keys here are not sorted. So it is an unordered lexicographic tree thus searching for an element would take longer time as compared to ordered hash tables. We will see this further in analysis section compairing CPU time of other data structures with Lexicographic Tree.
Another important thing to be noted is that it follows linear search. It means that one has to read whole tree in order to find an element. This can be improved by implementing binary search approach such as quick sort algorithm.
Marcin's lexicographic tree is similar to binary tree with more complexity because of linear search. It take complexity of O(n) where as binary tree with sorted list have complexity of O log(n) where n is the number of nodes.

5.2 Binary Tree vs Hash Tables

Hash tables in general have better cache behavior requiring less memory reads compared to a Binary Tree.
One advantage of binary tree over hash tables is that it is more dynamic. It do not reserve memory in advance or more than it might need. It can grow and shrink as required.
This is not true in case of hash tables. They are associative arrays and require static memory allocation(available slots to insert elements).They depend upon external hash functions used. Depending upon the range of hash functions, you need to allocate that amount of memory, even if you are going to hash just few number of elements.
For example: Bernstein hash allocates an array of 2000(pointers to) elements. Now if you are hashing just 100 elements, then rest of the memory space is wasted.
However, we used bigger datasets, it was just an example to illustrate difference between hash tables and binary trees.
Keep in mind that binary trees waste a lot of memory as well due to its linked implementation having pointers pointing to previous and next nodes. It stores at least two pointers for each element.
B trees:
B trees are generalization of binary trees.They are better than Binary trees.
Binary trees consist of nodes that have maximum two children. The first node is the parent and two children are called left and right.
B trees are used for insertion,search,deletion for larger datasets that require root node to have more than 2 leafs.Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. It is most commonly used in databases and filesystems.
Looking at the performance, hash tables are better than lexicographic tree.
Drawbacks can be avoid resizing of hash table by knowing exact number of elements to insert.
Hash tables are unordered . There is no sorted list mainly because it depends on hashing algorithm and elements are stored according to their hash value and not natural order.
It depends on our needs, which datastructure to use. If we do not need elements in their natural order, and we are aware of our dataset size(that it will not be growing any more), where we can avoid collisions by tweaking into hash functions and saving unused memory. Hash tables are good to go.
Hash tables can become complicated if dataset keeps growing in size because then we have to rehash elements,tweak into hash functions, reallocate memory according to table size.
If we need to do lots of inserts and few reads then hash table is good. But if we are doing less inserts and lots of reads then Btree is preferred.
B tree has expensive insert. [26]

5.3Design of the Benchmark

Operating System: Ubuntu 11.01 built over Linux Kernel, installed on Virtual Machine.
IDE: Eclipse Helios
Programming Language: c++
Remote Server: DIUFLX75, Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-29-generic x86_64)

A minimum of 26 million URIs were inserted in the data structure. Data type for key was an integer or unsigned integer depending upon the hash function used and data type for URI was String.
Reading data from the disk every single time will consume more memory and will not give precise results in benchmarking data structures on the basis of memory usage. To avoid this,data is loaded onto the RAM at once by adding it to a vector dynamically.
And then memory consumed by the process was measured. For each element,Key was computed by passing URI to a hash function and key-value pair was inserted in the data structure.Memory consumed by the process (VMdata) was measured again and subtracted the former memory from the latter to get the actual value of memory consumed by the data structure.CPU time and time taken to load data (in seconds) by the process were also measured. For incremental insert,data was fetched from the vector sequentially and thus noticed the difference in speed and memory as size of the data increased gradually. A project for each hash function and data structure pair was run 4 times and then average is computed for parameters such as CPU time and CPU cycles.

Results are further discussed in analysis section of the report.

5.4 Datasets

We experimented with four data sets. Each experiment was run 4 times to compute average of CPU time and CPU cycles.
Dataset 1 contained 26,28,8832 unique URIs extracted from dataset generated by Lehigh University Benchmark for 800 universities (approx- 26 Million).[30]
Dataset 2 contained 36,776,098 unique URIs extracted from dataset generated by BowlognaBench for160 departments (approx- 36 Million).[29]
Dataset 3 contained 52,616,588 unique URIs extracted from dataset generated by Lehigh University Benchmark for 1600 universities (approx- 52 Million)[30]
Dataset 4 contained 64,626,232 unique URIs extracted from dataset generated by DBpedia SPARQL Benchmark with scale factor of 200% (approx 64 Million)[28]

6 Computation of Results and Analysis

In upcoming sub sections of the report,we compute quality of data structures and hash functions individually on the basis of CPU time consumed and memory usage.

6.1Quality of Data Structures based on time and memory consumption

A data structure is said to be efficient if it performs faster and consumes less memory. We tested each data structure with different hash functions and measured time taken to insert data we experimented with all hash functions for each data structure and computed the results for CPU time , number of CPU cycles , memory consumed and number of collisions.

Table 6.1.1 Measurements for inserts in a Data Structure Std::Unordered_map with all hash functions for a datset of 26 Million URIs

Hf for ds std:unordered_map	CPU time (in seconds)	Number of CPU cycles	Number of collisions(in %)	Memory consumed(in gb)
Sbox	31.0	31030000	30.36	1.052
Jenkins	27.1	30570000	30.71	1.054
Murmur	25.6	28760000	0	1.055
Fnv	24.6	24660000	30.73	1.052
Paul	24.5	24510000	33.73	1.052
Bernstein	23.8	23860000	27.07	1.053
Mean	26.1	27231666.67	25.43	1.05

Table 6.1.2 Measurements for inserts in a Data Structure Gnu::hash_map with all hash functions for a datset of 26 Million URIs

Hf for ds Sgi/Gnu:hash_map	CPU time (in seconds)	Number of CPU cycles	Number of collisions(in %)	Memory consumed(in gb)
Sbox	25.0	25010000	30.36	1.156
Jenkins	22.8	22780000	30.71	1.155
Murmur	22.7	22370000	0	1.158
Fnv	19.6	19680000	30.73	1.155
Paul	19.4	19420000	33.73	1.155
Bernstein	19.3	19370000	27.07	1.156
Mean	21.47	21438333.33	25.43	1.16

Table 6.1.3 Measurements for inserts in a Data Structure TR1::Boost with all hash functions for a datset of 26 Million URIs

Hf for ds TR1::Boost	CPU time (in seconds)	Number of CPU cycles	Number of collisions(in %)	Memory consumed(in gb)
Sbox	34.8	34880000	30.36	1.156
Jenkins	31.2	37110000	30.71	1.155
Murmur	30.5	31220000	0	1.158
Fnv	27.8	27820000	30.73	1.155
Paul	27.0	27040000	33.73	1.155
Bernstein	25.7	19370000	27.07	1.156
Mean	29.5	29573333.33	25.43	1.16

Table 6.1.4 Measurements for inserts in a Data Structure Std::map with all hash functions for a datset of 26 Million URIs.

Hf for ds StdMap(ordered)	CPU time in seconds	Number of CPU cycles	Number of collisions(in %)	Memory consumed(in gb)
Sbox	53.2	53260000	30.36	1.56
Jenkins	49.0	49090000	30.71	1.562
Murmur	45.2	45210000	0	1.56
Fnv	44.2	45860000	30.73	1.562
Paul	45.8	19420000	33.73	1.561
Bernstein	26.4	26490000	27.07	1.563
Mean	43.97	39888333.33	25.43	1.56

We realised that hash function does not influence memory consumed by the data structure and type of data structure does not change number of collisions produced by a hash function.

Table 6.1.5 Measurements for inserts in a Data Structure Khash with all hash functions for a datset of 26 Million URIs.

Hf for ds Khash	CPU time (in seconds)	Number of CPU cycles	Number of collisions(in %)
Sbox	25.9	25960000	30.36
Jenkins	22.0	27020000	30.71
Murmur	19.4	22420000	0
Fnv	19.8	19860000	30.73
Paul	27.8	27820000	33.73
Bernstein	22.6	20670000	27.07
Memory Consumed by KHash	2.47 gb		

Table 6.1.6 Measurements for inserts in a Data Structure Sparse_Hash_Map with all hash functions for a datset of 26 Million URIs.

Hf for ds Sparse_Hash_Map	CPU time (in seconds)	Number of CPU cycles	Number of collisions (in %)
Sbox	140.6	148560000	30.36
Jenkins	139.5	139530000	30.71
Murmur	131.0	131030000	0
Fnv	93.87	102990000	30.73
Paul	130.6	130650000	33.73
Bernstein	90.5	93880000	27.07
Memory Consumed by Sparse_Hash_Map	0.42 gb		

Table 6.1.7 Comparison of data structures on the basis of speed(CPU time, CPU cycles) and memory consumed for 26 Million URIs Insert with Murmur Hash.

Data structure	**CPU time (in seconds)**	**Number of CPU cycles**	**Memory(in gb)**
Std:Unordered map	**26.1**	**27231666.67**	**1.05**
GNU::HashMap	**21.47**	**21438333.33**	**1.16**
K_hash	**22.78**	**23958333.33**	**2.47**
TR1::Boost	**29.5**	**29573333.33**	**1.16**
Std::Map(Ordered)	**43.97**	**39888333.33**	**1.56**
Sparse_Hash_Map	**121.01**	**124440000**	**0.42**
Lexicographic Tree	**35.48**	**35490000**	**4.89**

Our top three data structures in terms of speed are
GNU::HashMap, K_Hash,TR1::Unordered map
Our top three data structures in terms of least memory consumption are
Google:Sparse_Hash_Map,std:Unordered map and Gnu:hash map and TR1:Boost.

GNU Hash map consumes same amount of memory as TR1 Boost but it is faster in terms of speed.

Std::map takes more time. And it is understandable because it spends time in ordering. Our results agree with Nick Welch's work that Sparse hash map is 2-3 times slower but consumes half of the memory as compared to rest of the data structures

Note: uthash ran out of memory after 14 millions of data insert.
Since it didnot accomodate all the dataset. We did not consider it fit to use any further.

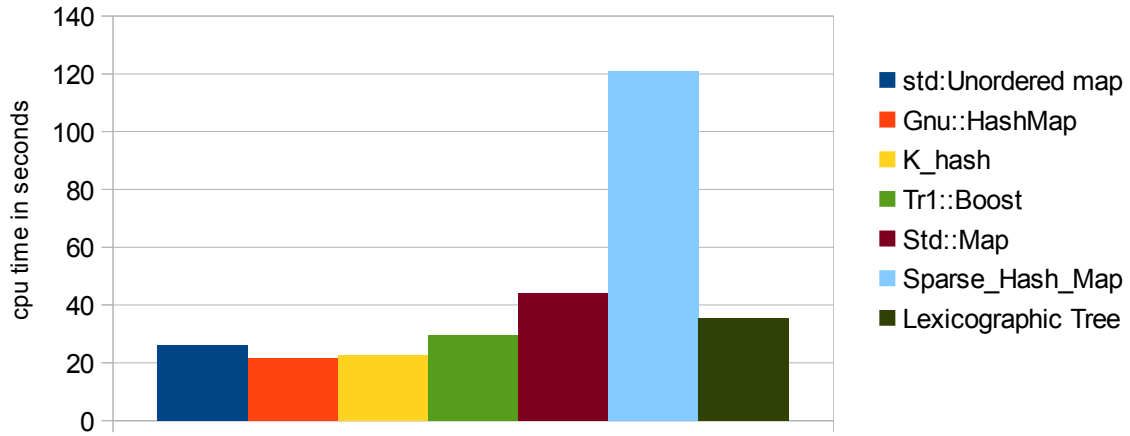

Figure:6.1.1 CPU time(in seconds) consumed by different data structures for 26 million URIs Insert with Murmur Hash.

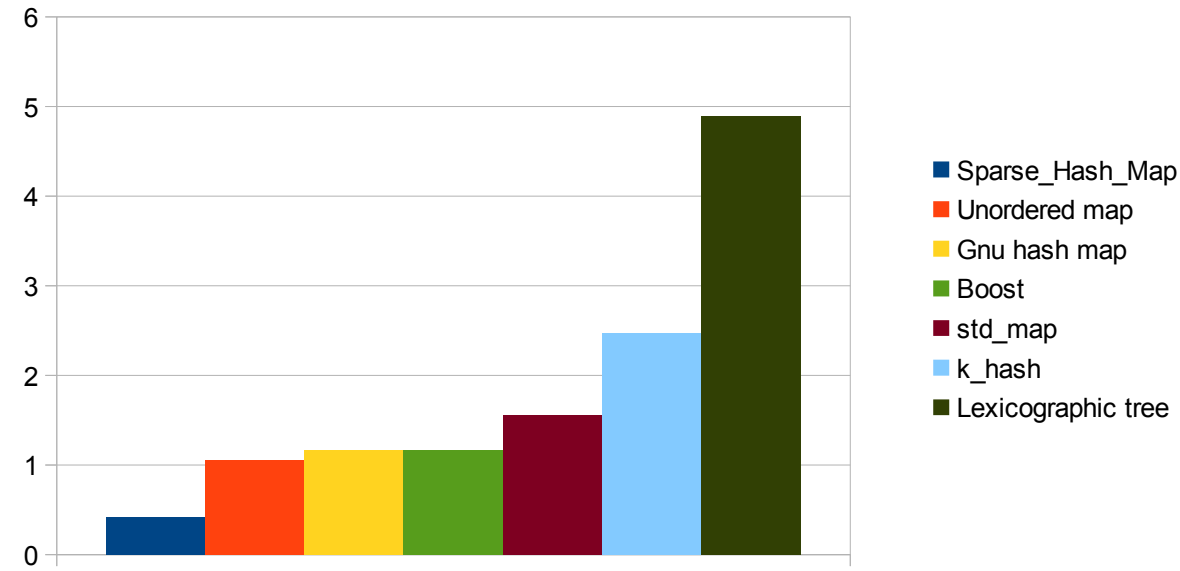

Figure:6.1.2 Memory Consumed(in gb) by different data structures for 26 million URIs Insert with Murmur Hash.

Table 6.1.8 Comparison of data structures on the basis of speed(CPU time, CPU cycles) and memory consumed for 36 Million URIs Inserts with Murmur Hash Function

Data structure	CPU time (in seconds)	Number of CPU cycles	Memory(in gb)
Std:Unordered map	45.5	36628000	1.65
Gnu::HashMap	36.0	36146000.6	1.47
K_hash	32.7	32677890.9	4.34
Tr1::Boost	53.52	53520000	1.47
Std::Map(Ordered)	87.1	87150000	2.19
Sparse_Hash_Map	170.9=2.0 minutes	170910000	0.61
Lexicographic Tree	61.9	61910000	7.43

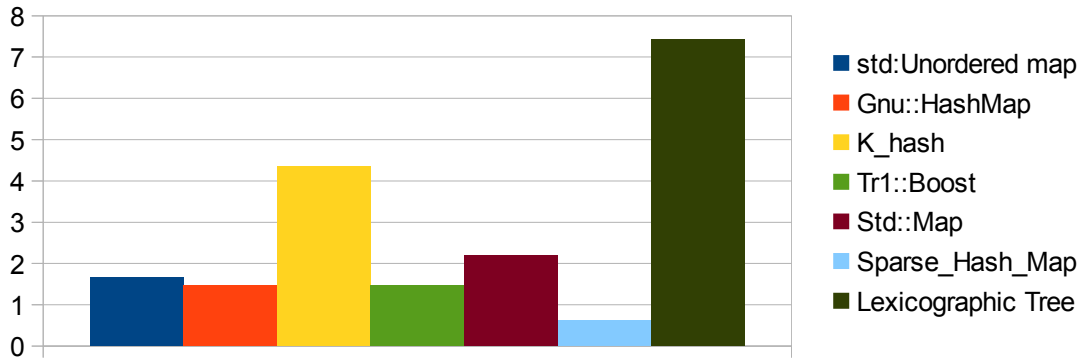

Figure:6.1.3 Memory Consumed(in gb) by different data structures for 36 million URIs Insert with Murmur Hash.

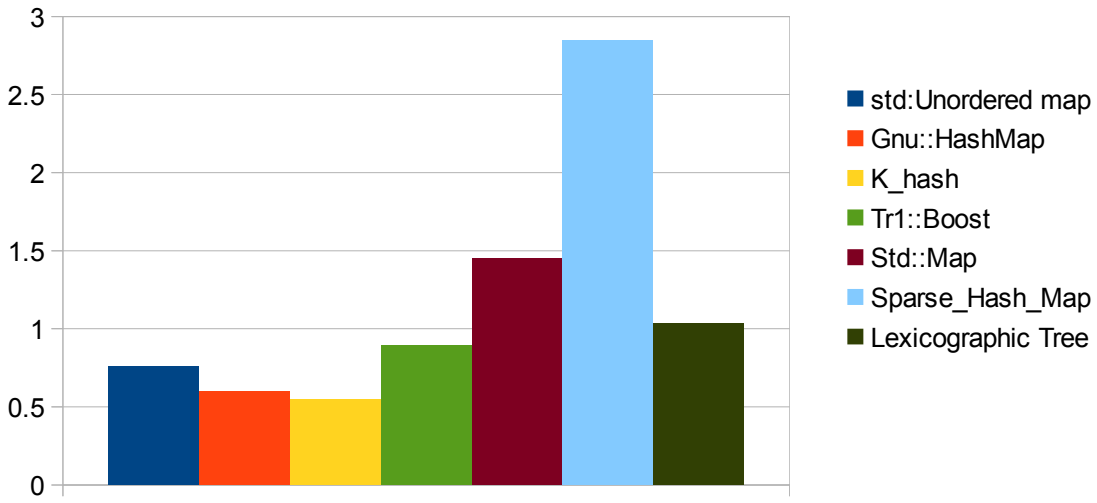

Figure:6.1.4 CPU time(in minutes) consumed by different data structures for 36 million URIs Insert with Murmur Hash.

Table 6.1.9 Comparison of data structures on the basis of speed(CPU time, CPU cycles) and memory consumed for 52 Million URIs Insert with Murmur Hash.

Data structure	CPU time (in seconds)	Number of CPU cycles	Memory(in gb)
Std:Unordered map	55.0	36628000	1.65
Gnu::HashMap	65.8	36146000.6	1.47
K_hash	38.8	32677890.9	4.34
Tr1:Boost	90.81	53520000	1.47
Std::Map(Ordered)	130.0	87150000	2.19
Sparse_Hash_Map	290.7=4.8 minutes	170910000	0.61
Lexicographic Tree	74.2	74290000	9.7

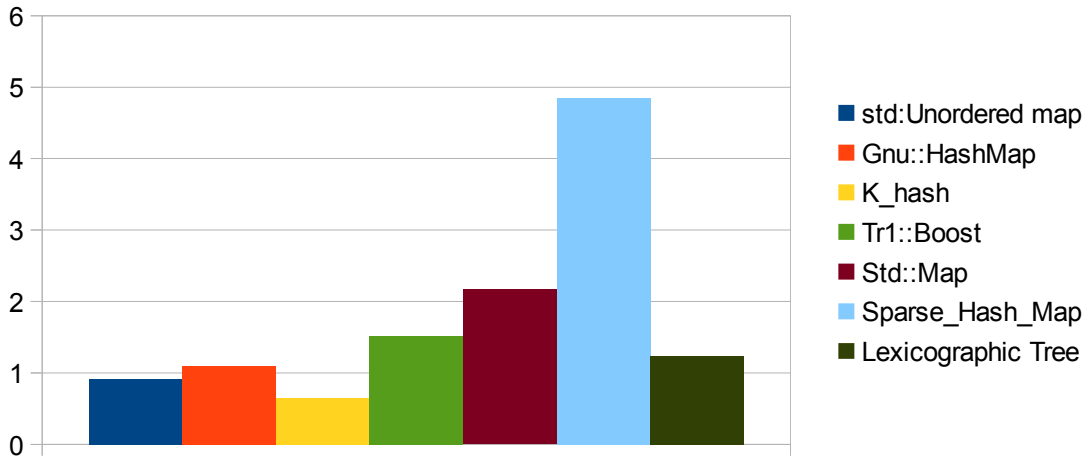

Figure:6.1.5 CPU time(in minutes) consumed by different data structures for 52 million URIs Insert with Murmur Hash.

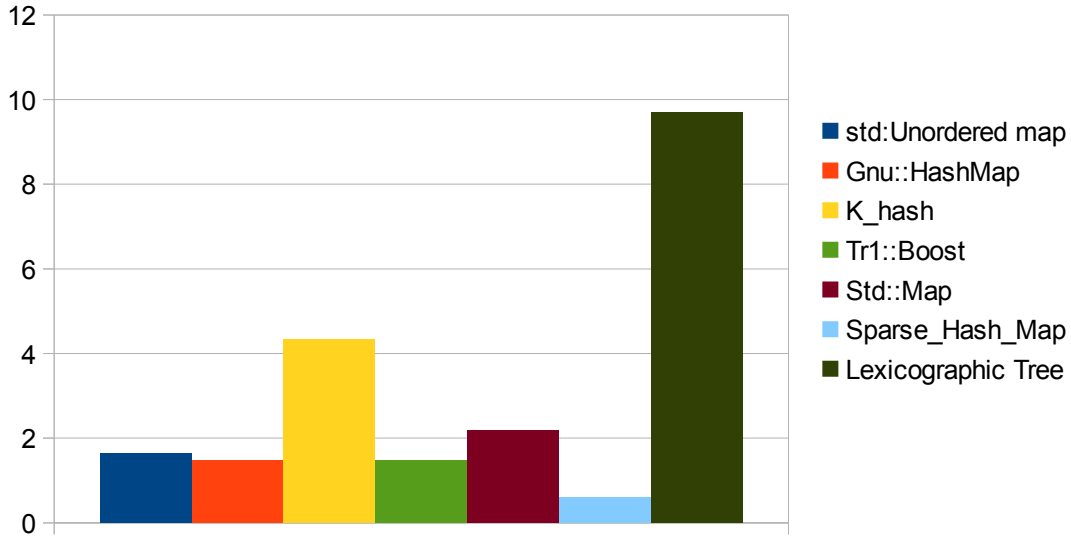

Figure:6.1.6 Memory Consumed(in gb)by different data structures for 52 million URIs Insert with Murmur Hash.

Table 6.1.10 Comparison of data structures on the basis of speed(CPU time, CPU cycles) and memory consumed for 64 Million URIs Insert with Murmur Hash.

Data structure	CPU time (in seconds)	Number of CPU cycles	Memory(in gb)
Std:Unordered map	60.0	59218493.2	2.48
Gnu::HashMap	70.0	72730000.5	2.68
K_hash	46.1	46160000	5.21
Tr1::Boost	105.3=1.7 minutes	105380000	2.68
Std::Map(Ordered)	140.4=2.3 minutes	149420000	3.85
Sparse_Hash_Map	387.3=6.4 minutes	388015000	1.08
Lexicographic Tree	212.6=3.5 minutes	212640000	14.5

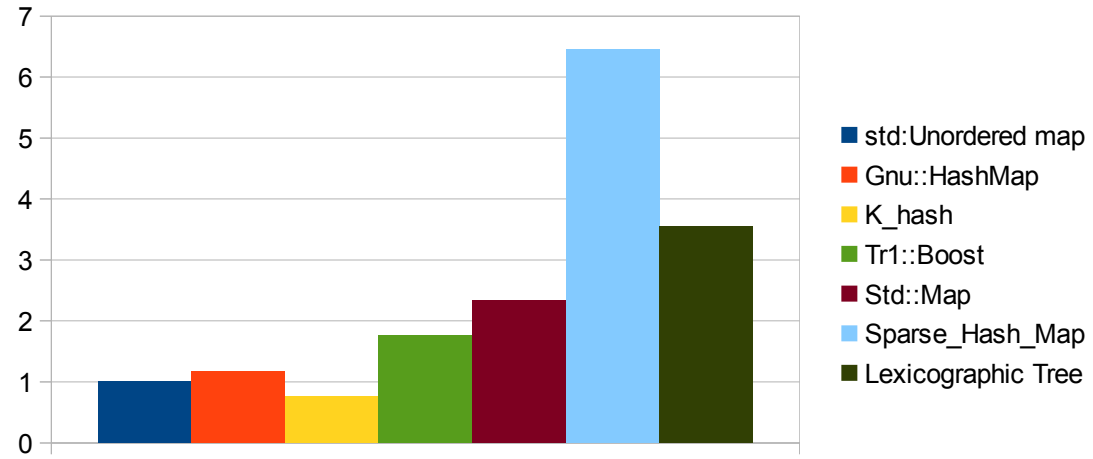

Figure:6.1.7 CPU time(in minutes) consumed by different data structures for 64 million URIs Insert with Murmur Hash.

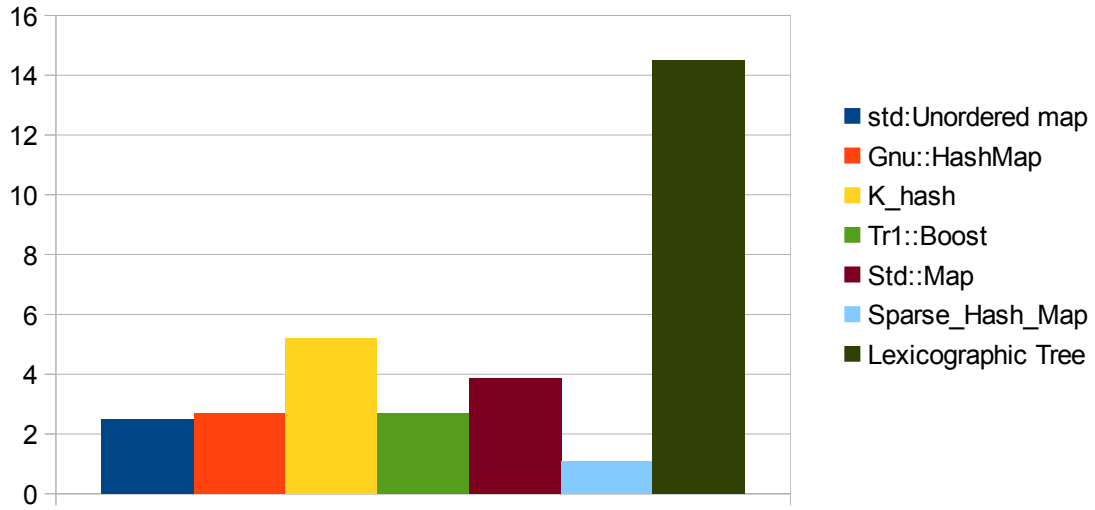

Figure:6.1.8 Memory Consumed(in gb)by different data structures for 64 million URIs Insert with Murmur Hash.

6.2Overview of Quality of Hash Functions

Strong hash functions are those that produce less number of collisions i.e. Same key for different strings and consumes less amount of time.

Usually good hash functions, having capacity to produce random keys for each element due to the use of strong mix of constants take more time.

For example Cryptographic hash functions such as MD5 and SHA-1 uses special purpose block ciphers for collision resistance making it difficult to produce same hash value for different messages. But they are generally slow in performance.

Table 6.2.1 Overview of Hash Functions based on speed and number of collisions for data set 1 , 26 million URIs

In each case, GNU:Hash Map was the data structure chosen since it is one of the fastest data structure in CPU time consumption.

Hash function	CPU time(in seconds)	Number of collision	% out of 26 million
Sbox	25.0	79817	0.30
Jenkins/One at time	22.3	80738	0.31
Murmur	22.0	0	0
Fnv	21.78	80784	0.31
Paul	20.3	88681	0.31
Bernstein	19.3	71159	0.27

Murmur hash function have least number of collisions followed by Bernstein and Sbox.

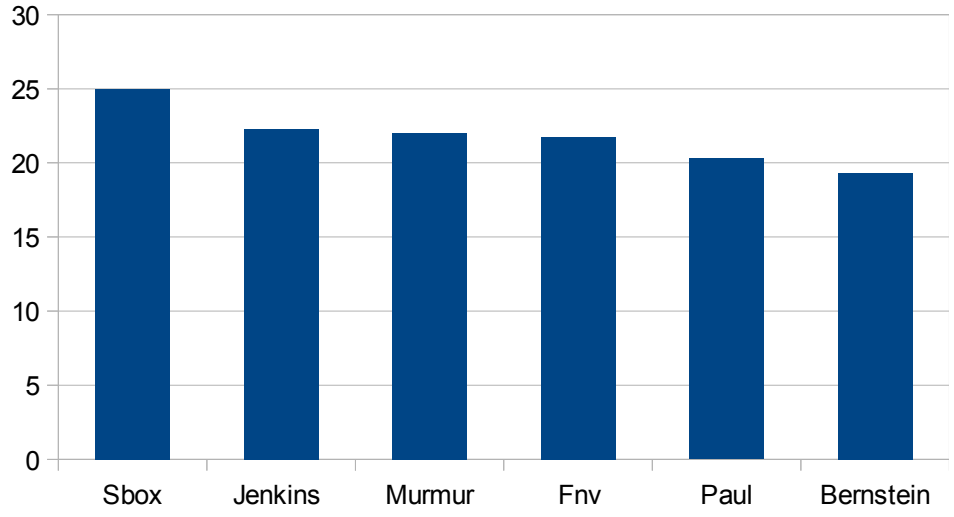

Figure 6.2.1: CPU time(in seconds) consumed by different hash functions for 26 million URIs Insert in GNU::Hash Map.

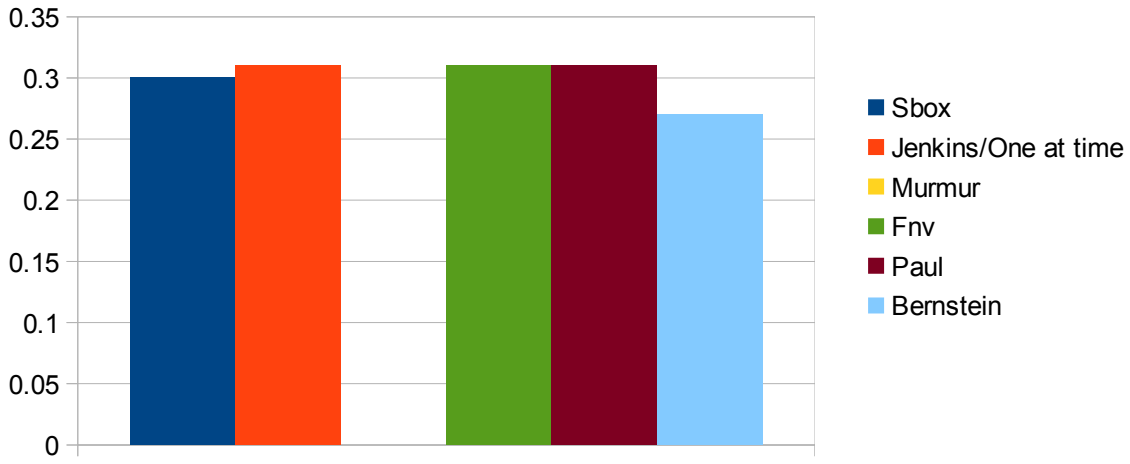

Figure 6.2.2:Number of Collisions (in percentage), produced by different hash functions for 26 million URIs Insert in GNU::Hash Map.

Table 6.2.2 Overview of Hash Functions based on speed and number of collisions for data set 2 ,36 million URIs with GNU::HashMap Data Structure.

Hash function	CPU time(in seconds)	No. of collisions	% out of 36 million
Sbox	36.0	156943	0.43
Jenkins	33.4	157505	0.43
Murmur	30.0	0	0
Fnv	35.2	156791	0.43
Paul	25.0	591302	1.61
Bernstein	36.6	155057	0.42

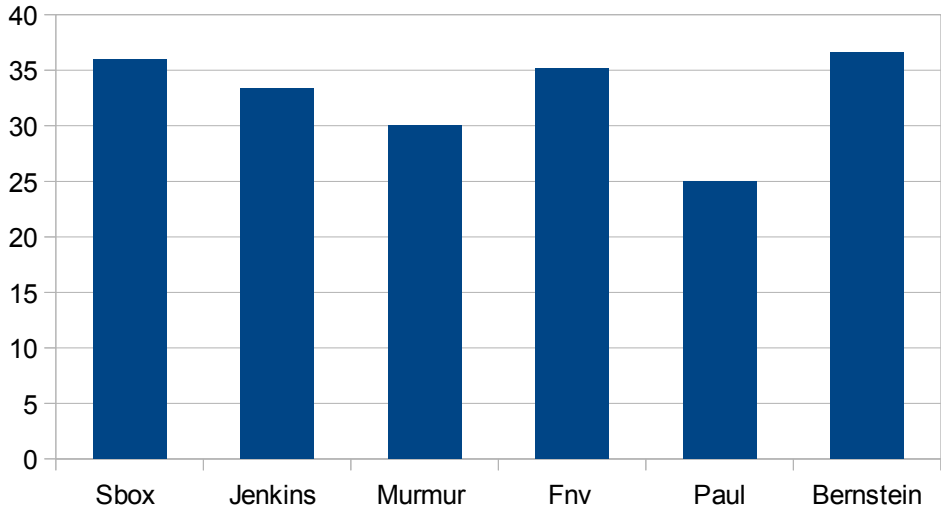

Figure 6.2.3: CPU time(in seconds) consumed by different hash functions for 36 million URIs Insert in GNU::Hash Map.

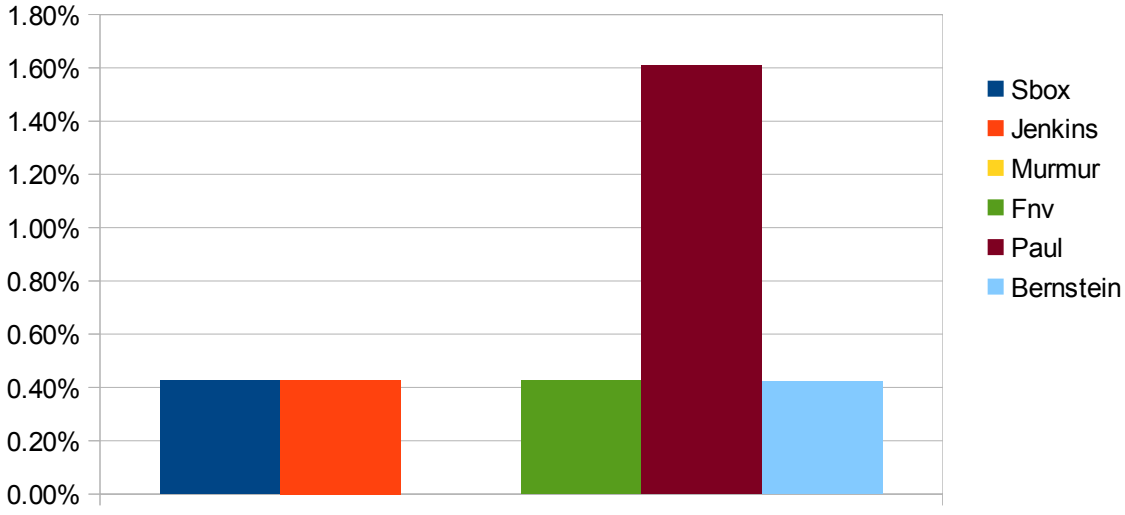

Figure 6.2.4:Number of Collisions in percentage, produced by different hash functions for 36 million URIs insert in GNU::Hash Map.

Table 6.2.3 Overview of hash functions based on speed and no of collisions for data set 3 ,52 million URIs with GNU::Hash Map Data Structure.

Hash function	CPU time(in seconds)	No. of collisions	% out of 52 million
Bernstein	50.6	304345	0.58
Fnv	42.6	322037	0.61
Paul	40.3	355243	0.68
Murmur	65.8	0	0
Jenkins	59.0	319013	0.61
Sbox	64.3	321505	0.61

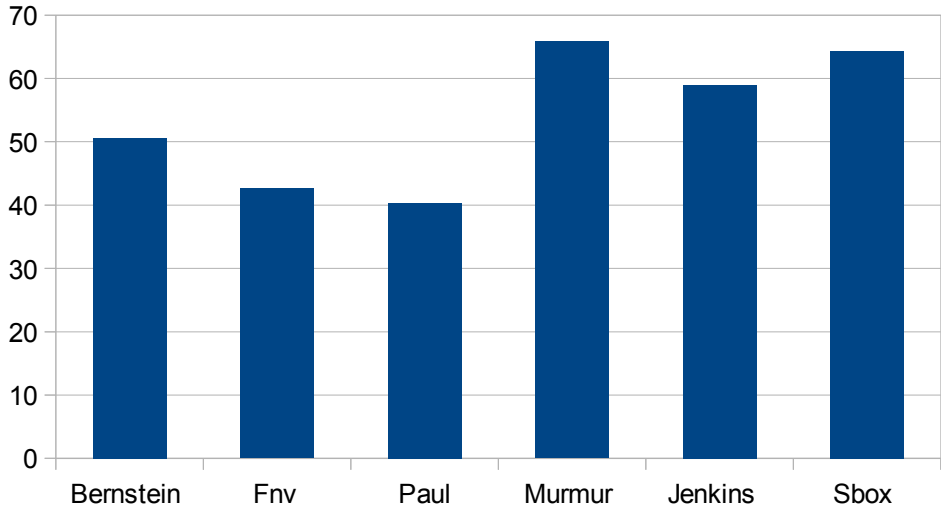

Figure 6.2.5: CPU time(in seconds) consumed by different hash functions for 52 million URIs Insert in GNU::Hash Map.

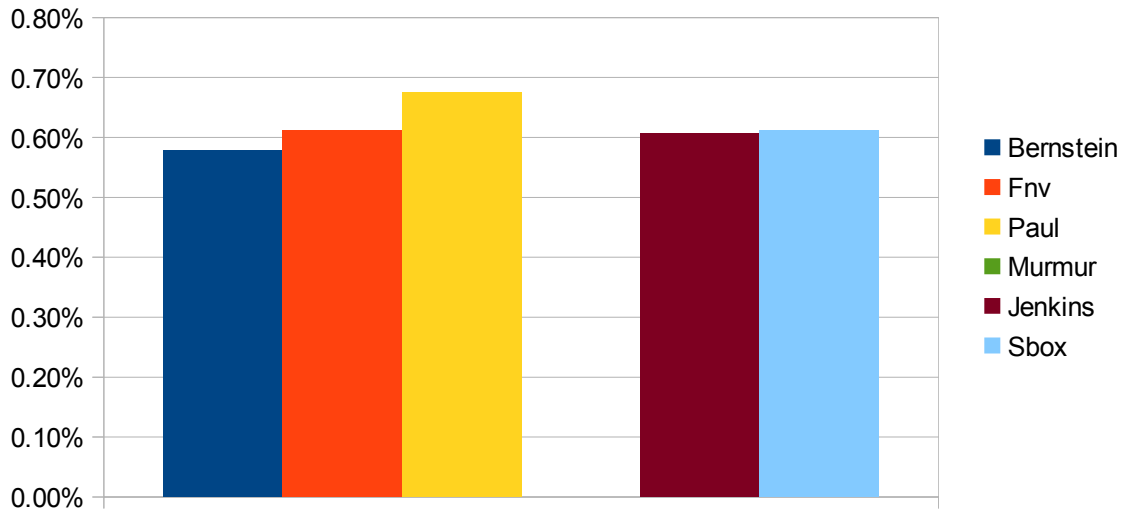

Figure 6.2.6:Number of Collisions in percentage, produced by different hash functions for 52 million URIs Insert in GNU::Hash Map.

Table 6.2.4 Overview of hash functions based on speed and no of collisions for data set 4 ,64 million URIs with GNU::Hash Map Data Structure.

Hash function	Cpu time(in seconds)	No. of collisions	% out of 64 million
Bernstein	71.3	579078	0.90
Fnv	63.9	481584	0.75
Paul	59.0	7144887	11.06
Murmur	70.0	0	0
Jenkins	56.7	488461	0.76
Sbox	73.5	483115	0.75

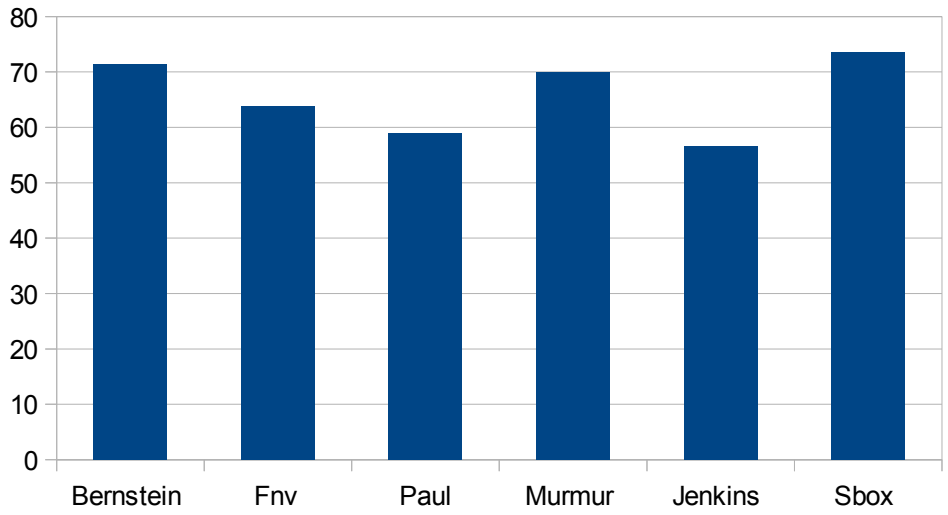

Figure 6.2.7: CPU time(in seconds) consumed by different hash functions for 64 million URIs Insert in GNU::Hash Map.

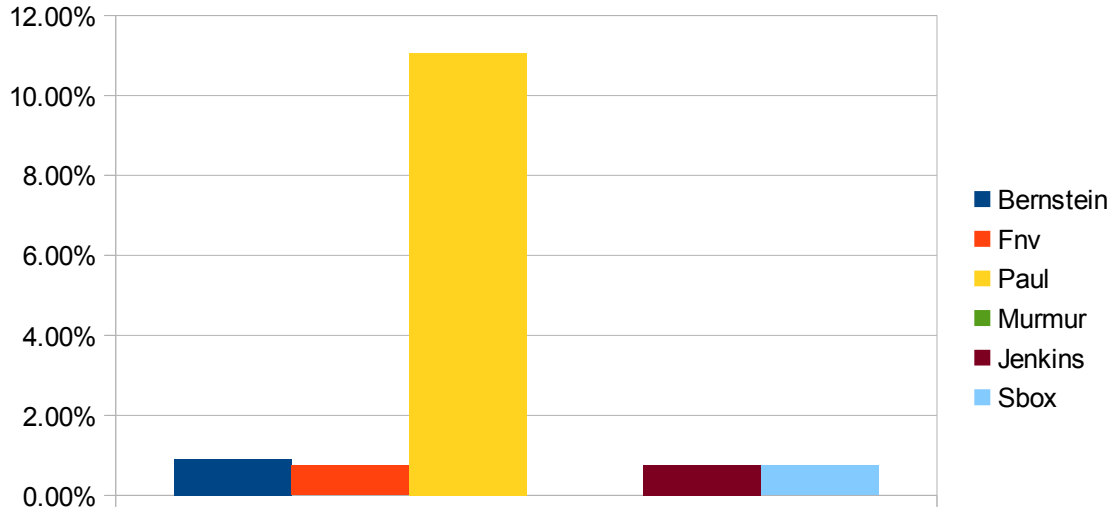

Figure 6.2.8:Number of Collisions in percentage, produced by different hash functions for 64 million URIs Insert in GNU::Hash Map.

7 INCREMENTAL INSERTS

For every 5 million URIs insertion, cpu time and memory consumed by the data structure were noticed. This experiment was conducted to see the difference between time and memory as data set grew in size. The results are not cumulative.Instead they represent for every next 5 million inserts by subtracting the former value.

Split – up

5,000,000= Five Million
10,000,000= Ten Million
15,000,000= Fifteen Million
20,000,000= Twenty Million
25, 000,000= Twenty Five Million
30,000,000= Thirty Million
35,000,000= Thirty Five Million
40,000,000= Forty Million
45,000,000= Forty Five Million
50,000,000= Fifty Million
55,000,000= Fifty Five Million
60,000,000= Sixty Million

Experiment was conducted for top two hash functions. They were:
Murmur hash with 0 collision (consumes more time as compared to other hash functions) and Bernstein Hash with least number of collisions (comparatively faster meaning consumes less time as compared to Murmur)

Table 7.1 CPU time (in minutes) for each data structure with Murmur hash for every 5 million elements up to 60 Million inserts.

inserts in Millions	Std::Map	Unorderedmap	Boost	HashMap	SparseHash	LexicoTree	K_Hash
5M	0.14	0.08	0.05	0.05	0.26	0.15	0.06
10M	0.32	0.17	0.11	0.11	0.56	0.3	0.13
15M	0.52	0.21	0.2	0.19	0.95	0.45	0.17
20M	0.74	0.36	0.23	0.22	1.22	0.59	0.28
25M	0.98	0.4	0.27	0.26	1.56	0.85	0.33
30M	1.22	0.44	0.42	0.43	2.06	1.16	0.39
35M	1.47	0.49	0.45	0.47	2.34	1.48	0.45
40M	1.73	0.78	0.49	0.51	2.65	1.82	0.63
45M	1.99	0.83	0.53	0.55	3	2.18	0.69
50M	2.26	0.87	0.57	0.6	3.38	2.58	0.75
55M	2.54	0.92	0.86	0.96	4.18	2.88	0.82
60M	2.82	0.97	0.9	1.01	4.45	3.36	0.89

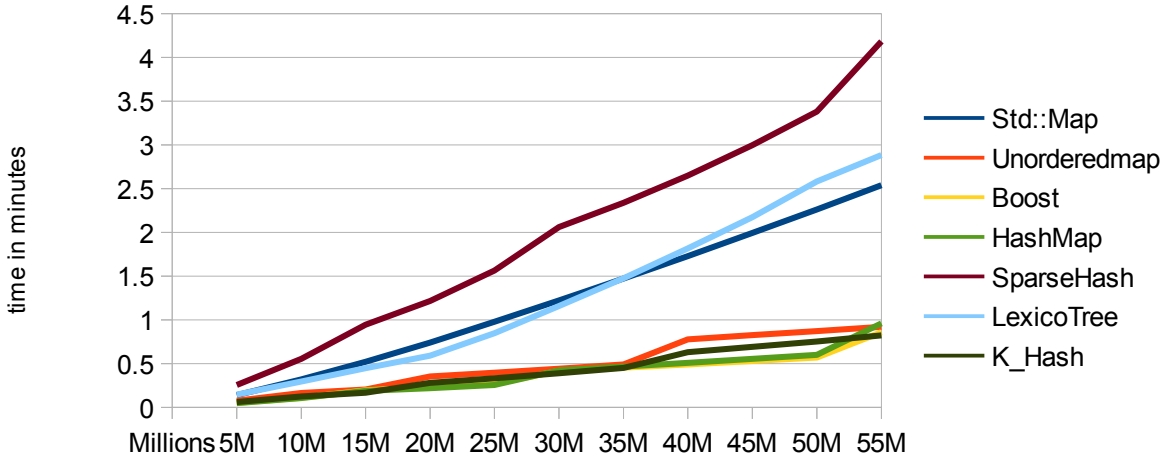

Figure7.1 Chart for CPU time (in minutes) for each data structure with Murmur hash for every 5 million elements up to 60 Million inserts.

Table 7.2 CPU time (in minutes) for each data structure with Bernstein hash for every 5 million elements up to 60 Million inserts.

inserts	Std::Map	Unorderedmap	Boost	HashMap	SparseHash	LexicoTree	K_Hash
in Millions							
5M	0.12	0.07	0.07	0.07	0.26	0.15	0.07
10M	0.27	0.15	0.15	0.15	0.56	0.3	0.16
15M	0.43	0.2	0.24	0.24	0.94	0.45	0.21
20M	0.59	0.31	0.3	0.3	1.19	0.59	0.32
25M	0.75	0.35	0.36	0.36	1.53	0.85	0.38
30M	0.94	0.4	0.51	0.51	2.05	1.16	0.44
35M	1.12	0.46	0.57	0.57	2.33	1.48	0.51
40M	1.31	0.65	0.63	0.63	2.64	1.82	0.67
45M	1.51	0.7	0.69	0.69	2.98	2.18	0.73
50M	1.71	0.75	0.76	0.76	3.36	2.58	0.79
55M	1.91	0.8	1.04	1.04	4.18	2.88	0.87
60M	2.12	0.85	1.1	1.1	4.45	3.36	0.94

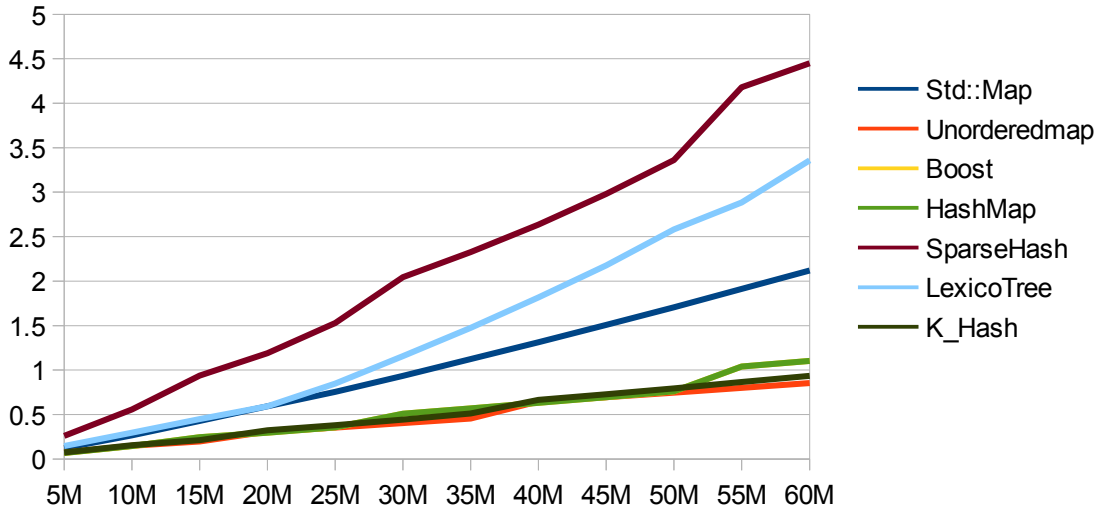

Figure7.2 Chart for CPU time (in minutes) for each data structure with Bernstein hash for every 5 million elements up to 60 Million inserts.

Table 7.3 CPU time (in minutes) for each Hash Function with gnu:HashMap(one of the fastest data structure) for every 5 million elements up to 60 Million inserts.

inserts in Millions	Murmur	Bernstein	Jenkins Hash	Paul Hash	FNV Hash	S Box
5M	0.05	0.07	0.05	0.04	0.06	0.06
10M	0.11	0.15	0.11	0.09	0.13	0.13
15M	0.19	0.24	0.19	0.13	0.22	0.22
20M	0.22	0.3	0.23	0.19	0.26	0.27
25M	0.26	0.36	0.27	0.23	0.31	0.33
30M	0.43	0.51	0.4	0.26	0.45	0.45
35M	0.47	0.57	0.44	0.37	0.5	0.51
40M	0.51	0.63	0.5	0.41	0.55	0.57
45M	0.55	0.69	0.55	0.44	0.6	0.63
50M	0.6	0.76	0.6	0.48	0.66	0.68
55M	0.96	1.04	0.85	0.52	0.93	0.9
60M	1.01	1.1	0.9	0.72	0.98	0.96

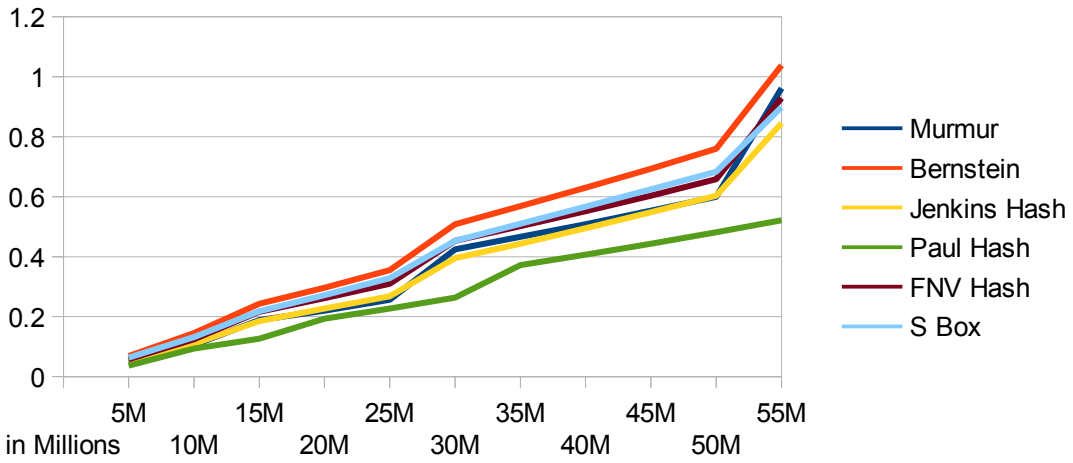

Figure7.3 Chart for CPU time (in minutes) for each Hash Function with gnu:HashMap(one of the fastest data structure) for every 5 million elements up to 60 Million inserts.

Table 7.4 Number of collisions (in percentage) for each Hash Function for every 5 million elements up to 60 Million inserts.

inserts in Millions	Murmur	Bernstein	Jenkins Hash	Paul Hash	FNV Hash	S Box
5M	0.00%	0.00%	0.01%	3.04%	0.00%	0.00%
10M	0.00%	0.01%	0.02%	5.68%	0.02%	0.02%
15M	0.00%	0.04%	0.04%	8.01%	0.04%	0.04%
20M	0.00%	0.07%	0.08%	10.30%	0.07%	0.07%
25M	0.00%	0.14%	0.12%	10.38%	0.11%	0.11%
30M	0.00%	0.21%	0.17%	10.44%	0.16%	0.16%
35M	0.00%	0.28%	0.22%	10.49%	0.22%	0.22%
40M	0.00%	0.36%	0.29%	10.56%	0.28%	0.29%
45M	0.00%	0.45%	0.37%	10.64%	0.36%	0.36%
50M	0.00%	0.55%	0.45%	10.73%	0.45%	0.45%
55M	0.00%	0.65%	0.55%	10.83%	0.54%	0.54%
60M	0.00%	0.75%	0.65%	10.95%	0.64%	0.64%

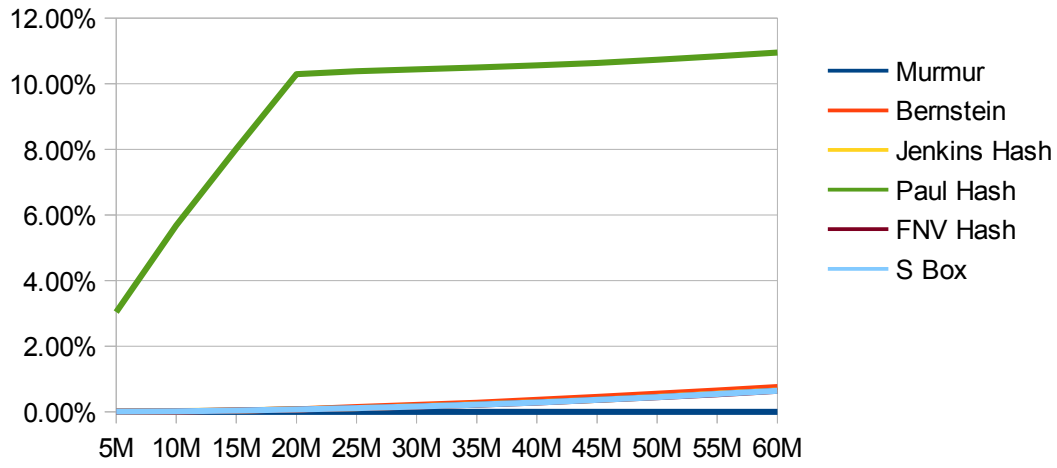

Figure7.4 Chart showing number of collisions (in percentage) for each Hash Function for every 5 million elements up to 60 Million inserts.

Table7.5 Memory consumed (in gb) by each data structure for every 5 million elements up to 60 Million inserts.

inserts in Millions	Std::Map	Unorderedmap	Boost	HashMap	SparseHash	LexicoTree	K_Hash
5M	0.30	0.21	0.20	0.20	0.08	1.15	0.44
10M	0.60	0.43	0.39	0.39	0.16	2.31	0.88
15M	0.89	0.58	0.63	0.63	0.25	3.49	1.17
20M	1.19	0.87	0.78	0.78	0.33	4.64	1.75
25M	1.49	1.01	0.93	0.93	0.41	5.72	2.02
30M	1.78	1.16	1.27	1.27	0.51	6.83	2.34
35M	2.08	1.31	1.41	1.41	0.58	7.94	2.67
40M	2.37	1.74	1.56	1.56	0.66	9.05	3.87
45M	2.67	1.89	1.71	1.71	0.73	10.16	3.87
50M	2.96	2.03	1.86	1.86	0.81	11.27	4.19
55M	3.26	2.18	2.38	2.38	0.93	12.40	4.58
60M	3.56	2.33	2.53	2.53	1.01	13.51	4.94

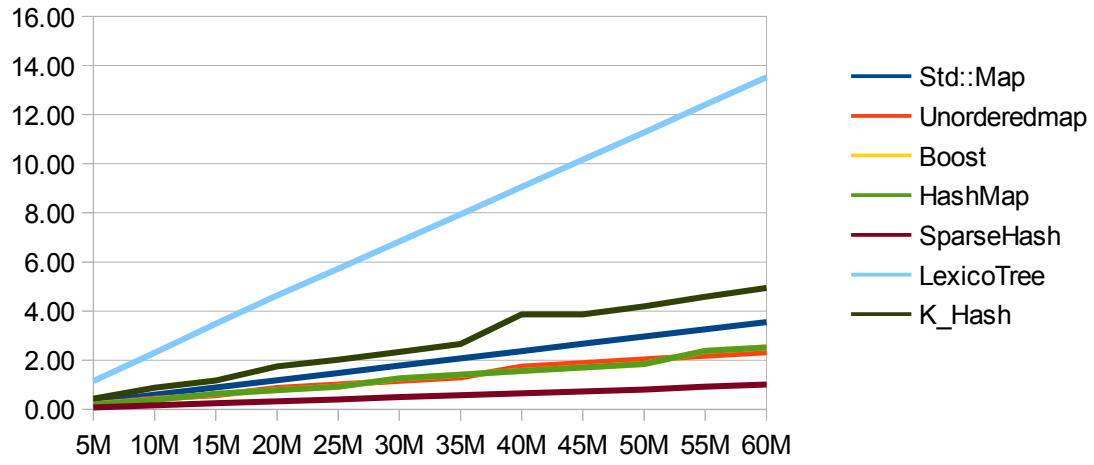

Figure7.5Chart showing memory consumed (in gb) by each data structure for every 5 million elements up to 60 Million inserts.

8 Computation of CPU time during Retrieval of Data

10 percent of the largest data set(64 million) that is approx 6 million elements were retrieved by giving URI and CPU time was noticed.

Table 8.1 CPU time during retrieval of 6 million elements by URI

Data structure with Murmur Hash	CPU time(in seconds) during retrieval of 6 Million elements by retrieving URI
Gnu:Hash Map	3.85
Std:Unordered map	3.8
Std:map	7.7
Sparse hash map	18.6
Khash	3.4
Tr1:Boost	4.4
Lexicographic Tree	10.6

Table 8.2 CPU time during retrieval of 6 million elements by ID (key)

Hash function with GNU:: Hash Map	CPU time(in seconds) when key is retrieved
Paul	3.47
Murmur hash	3.85
Sbox	4.3
Fnv	3.2
Jenkins	4.6
Bernstein	3.5
Lexicographic Tree	10.6

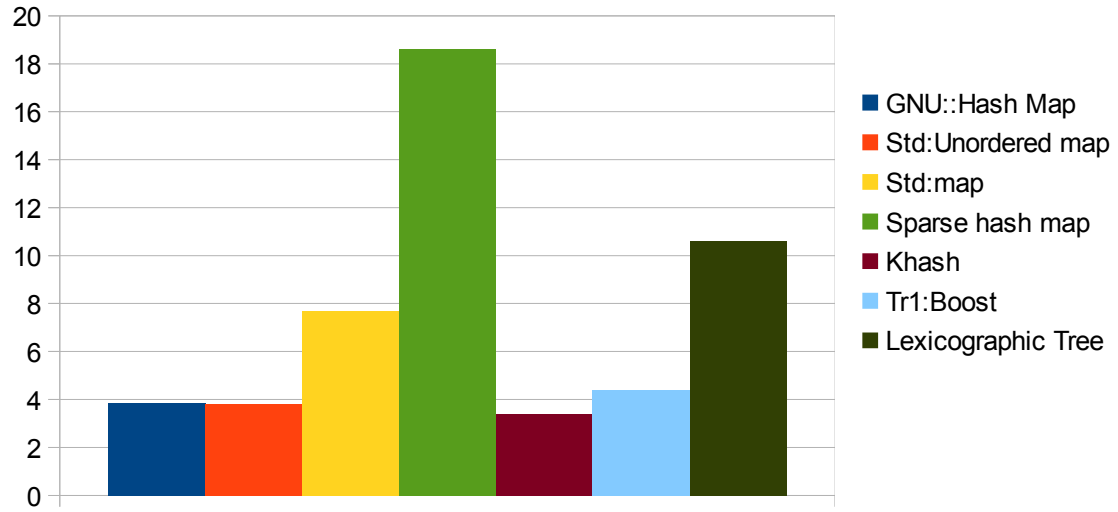

Figure8.1Chart showing CPU time (in seconds) during retrieval of 6 million elements by retrieving URI.

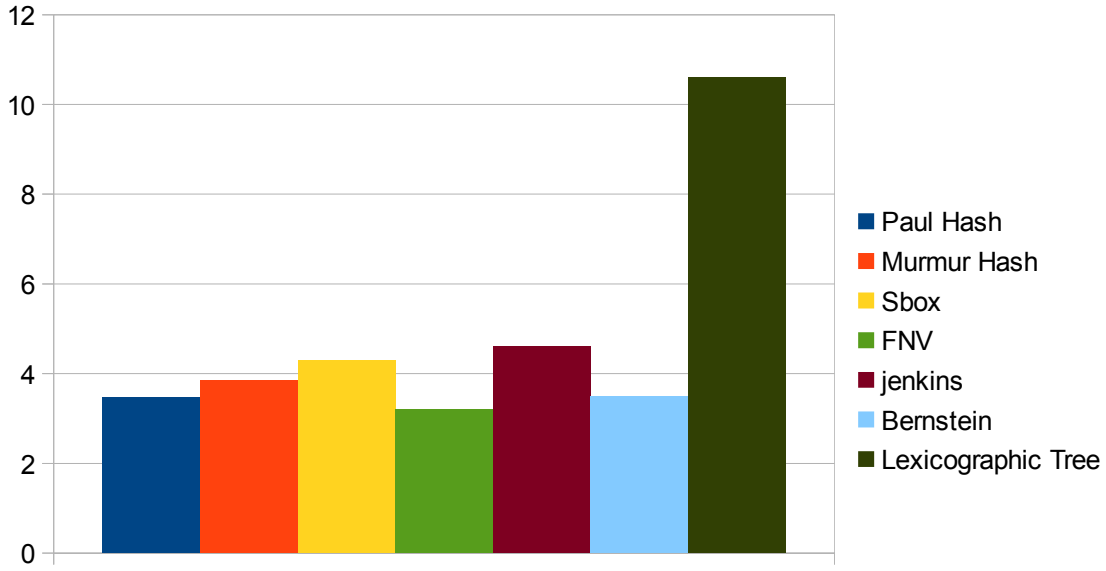

Figure8.2Chart showing CPU time(in seconds) during retrieval of 6 million elements by ID(key).

9 Future Work

Due to time constraint, we could not benchmark RDF-3X(RDF Triple eXpress). It is designed and implemented from scratch for the management and querying RDF Data. It is developed by Thomas Neumann at Max Planck Center, Germany.
RDF-3X is based on three basic principles:
1."Physical design is workload-independent by creating appropriate indexes over a single giant triples table.
2.The query processor is RISC-style by relying mostly on merge joins over sorted index lists.
3.The query optimizer mostly focuses on join order in its generation of execution plans." [32]
It will also be interesting to benchmark cryptographic hash functions such as MD 5,SHA-1 and compare them with non- cyrptographic hash functions, benchmarked in this report. They are usually slow in performance but they are collision resistant.

10 Conclusion

Lexicographic Tree consumes highest amount of memory as compared to other data structures.(14 gb for 64 Million URIs). It also consumes considerable amount of CPU time and thus I place it at POSITION 7.(Last position)
Sparse hash map consumes least amount of memory for all datasets. But it is the slowest among all data structures. Difference between its CPU time and Lexicographic Tree's CPU time is significantly low(2.9 seconds for 64 Million URIs)as compared to their memory consumption difference i.e. 13.4 gb thus Sparse hash map overall performs better than lexicographic tree and I place it at POSITION 6.
K_hash is faster than std_map but consumes more memory. It consumes 1.3 gb more as compared to std_map and std map consumes1.5 seconds more as compared to K_hash for 64 Million URIs. Since we are more senstive to memory rather than speed, I will chose k_hash at POSITION 5.
std map (ordered map) takes more time because of ordering elements. I place it at POSITION 4
TR1::Boost consumes same amount of memory as GNU:Hash Map but it takes more amount of CPU time. Thus it is placed at POSITION 3
GNU::Hash Map is faster than TR1:Boost and thus is placed at POSITION 2
Std:unordered map is fastest(60 seconds) and consumes least amount of memory(2.48) for 64 million URIs and thus I place it at POSITION 1.
Note: GNU hash map and std:Unordered map are very close in numbers. Infact for 52 Million data set , GNU hash map performed better than std::Unordered map in terms of speed and memory.
Paul claimed that his hash function, is around 66 % faster as compared to Jenkin's One at a Time Hash. Our experiment confirms it as well. But overall performance of a strong hash function depends upon its speed and number of collisions produced.
Paul's Super Fast Hash is actually faster (consumes less cpu time) as compared to Jenkins Hash for all datasets varying from 26 million to 64 million elements.
But it also produces more collisions as dataset grew larger in size. Infact it produced maximum number of collisions(11%) in the largest dataset of around 64 Million URIs as compared to other hash functions. Thus I place it to the last level(LEVEL 6).
Sbox was worst for 26 million inserts but as dataset grew in size, it started performing better in terms of number of collisions and cpu time didnot increase too much. It consumes more

time as compared to Paul Hash but it produces less amount of collisions. Difference in their number of collisions is far huge than difference in their CPU time consumed. Thus I place it to LEVEL 5.
Jenkins hash produced almost same amount of collisions as Sbox hash but it was significantly faster thus I place it to LEVEL 4 .
Fnv is faster than Jenkins hash and produced almost same percentage of collisions as Jenkins in all the datasets. Thus I place fnv at LEVEL 3
Bernstein Hash in doubt produced less number of collisions compared to other hash functions and also consumed less CPU time as compared to rest of them. Thus I place it at LEVEL 2
Murmur Hash is the best hash function among all and proudly takes LEVEL 1. It produced NO collision for all the datasets. It consumed considerable amount of CPU time but it is logical to understand that powerful hash function producing no collision will be slower. Nevertheless, it did not consume significantly huge time. It was just 20-25 seconds slower as compared to Paul Hash which produced maximum number of collisions.

Table 10.1 Ranking of Data Structures and Hash Functions

Level/Position	Data Structure	Hash Function
1	Std:Unordered map	Murmur Hash
2	Gnu::HashMap	Bernstein Hash
3	Tr1::Boost	FNV Hash
4	Std::map	Jenkins Hash
5	K_Hash	Sbox Hash
6	Sparse_Hash_Map	Paul Hash
7	Lexicographic Tree	-

Appendix A
Common Acronyms

HF =Hash function
DS = Data structure
RDF= resource data framework
URI= Uniform Resource Identifier
IDE= Integrated Development Environment
Approx=Approximately
no.=Number

Common Synonyms

Gnu:HashMap=Sgi:HashMap
Boost=Tr1:Boost
Google:Sparse_Hash_Map=Sparse_Hash_Map
Std:UnorderedMap=UnorderedMap
Cpu usage=Memory
Cpu Time=Time=Speed
One at a Time Hash=Jenkin's Hash
Paul Hash= Super Fast Hash

Appendix B
License of the Documentation

References

[1]R.Sint,S.Schaffert,S.Stroka,R.Ferstl,"Combining Unstructured, fully Strucutred and Semi Structured Information in semantic Wikis" http://stexx.files.wordpress.com/2010/07/combining_unstructured_semistructured_fullystructured_data.pdf (accessed November 20,2012)
[2]Ian davis,"30 Minute Guide to Rdf and LinkedData" http://www.slideshare.net/iandavis/30-minute-guide-to-rdf-and-linked-data (accessed November 20,2012)
[3]Linux Programmer's Manual http://www.kernel.org/doc/manpages/online/pages/man5/proc.5.html (accessed November 20,2012)
[4] getrusage(2) Linux man page http://linux.die.net/man/2/getrusage(accessed December 2012)
[5] Elapsed Time http://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html(accessed December 2012)
[6] Clock Cycle http://www.techterms.com/definition/clockcycle(accessed December 2012)
[7]Performance Application Programming Interfacehttp://icl.cs.utk.edu/PAPI/
[8] Data Structures and its Types http://www.slideshare.net/NavtarSidhuBrar/data-structure-and-its-types-7577762
[9] sparse_hash_map http://goog-sparsehash.sourceforge.net/doc/sparse_hash_map.html(accessed December 2012)
[10]sgi, hash_map http://www.sgi.com/tech/stl/hash_map.html(accessed December 2012)
[11] cplusplus, std::map available: http://www.cplusplus.com/reference/map/map/ (accessed December 2012)
[12]std::unordered_map http://en.cppreference.com/w/cpp/container/unordered_map(accessed December 2012)
[13] Boost.Unordered http://www.boost.org/doc/libs/1_37_0/doc/html/unordered.html(accessed December 2012)
[14]uthash user guide http://troydhanson.github.com/uthash/userguide.html(accessed December 2012)
[15] Implementing Generic Hash Library in C http://attractivechaos.wordpress.com/2008/09/02/implementing-generic-hash-library-in-c/ (accessed December 2012)
[16]I,Enchev,"Unconventional Store Systems for RDF Data"(accessed January 2013)
[17] Free Documentation Licence (GNU FDL) http://www.gnu.org/licenses/fdl.txt (accessed January 10, 2013).
[18]Hash Functions http://home.comcast.net/~bretm/hash/(accessed January 2013)
[19]Hash functions : An emperical Comparison http://www.strchr.com/hash_functions?allcomments=1#comment_363(accessed January 2013)
[20]FNV Hash http://www.isthe.com/chongo/tech/comp/fnv/(accessed January 2013)
[21]A,Appleby,"Murmur hash"available: https://sites.google.com/site/murmurhash/ (accessed December 2012)
[22]The Use of Substitution Boxes in HashFunctions http://home.comcast.net/~bretm/hash/10.html(accessed December 2012)

[23] J.Walker “Eternally Confuzzled” http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx(accessed December 2012)
[24]P. Hsieh,”Hash Functions” http://www.azillionmonkeys.com/qed/hash.html(accessed December 2012)
[25] Hash Functions(Continued),Mixing Functions http://home.comcast.net/~bretm/hash/3.html(accessed December 2012)
[26]S,Haines,The Difference between Trees and Hash Tables http://www.informit.com/guides/content.aspx?g=java&seqNum=472(accessed December 2012)
[27]N.Welch,Incise.org,Hash Table Benchmarks.Available: http://incise.org/hash-table-benchmarks.html(accessed January 2013)
[28]M.Morsey,J.Lehmann,S.Auer,A.C.NyongaNgomo.”DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data”(accessed February 2013)
[29]G.Demartini,I.Enchev,M.Wylot,J.E.Gapany,P.C.Mauroux,”BowlognaBench-Benchmarking RDF Analytics"(accessed February 2013)
[30]Guo, Yuanbo, Pan, Zhengxiang and Heflin, Jeff . LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics. 3(2) July 2005.pp.158-182.
[31]M.Wylot,J.E Pont,M.Wisniewski,P.C-Mauroux,”dipLODocus[RDF] - Short and Long-Tail RDF Analytics for Massive Webs of Data",ISWC2011(accessed March 2013)
[32]T.Neumann,G.Weikum,”The RDF-3X Engine for Scalable Management of RDF Data” (accessed March 2013)