Technical Report of iZENELib

Yingfeng Zhang, Kevin Hu

December 12, 2008

Abstract

This document presents the technical report for the project *iZENELib* that could be used in the search engine developing process. *iZENELib* is expected to contain two parts:AM-Lib, which takes charge of storage, and IR-Lib, which takes charge of information retrieval and machine learning.

Contents

1 Document History

Date	Author	Description
2008-11-02	Kevin	Initialize the design issue of AM-Lib.
2008-11-10	Yingfeng	Initialize the design issue of IR-Lib.
2008-11-10	Yingfeng	Create the technical report.
2008-11-21	Yingfeng	Add the design issue of Matrix Library.
2008-12-05	Yingfeng	Adjust the milestone.

2 Design Goal

iZENELib plans to provide a collection of utilities which could be used in the search engine developing process. *iZENELib* is expected to be composed of two parts—the part taking charge of storage(AM-Lib), and the part in charge of information retrieval and machine learning(IR-Lib). AM-Lib is expected to be composed of two sub-parts, the one which provides common utilities for data storage, and the one providing corpus data management which serves for the IR-Lib. Generic design would be adopted largely in *iZENELib* to provide much more flexibility for component's reusing.

3 Initial architecture design of AM-lib

The future probable usages of AM-lib can be illustrated in several ways.

- In your project, you may want to compare the performance of using several different data structures. AM-lib makes this easy anyway.

```
#include "am/am.hpp’’
class MyClass {

public:
        MyClass(AccessMethods<int, string>* pAm): pAm_(pAm){}

        void myFoo()
        {
                pAm_->insert(128, ‘‘Hello! AM-lib‘‘);
                pAm_->getDataBy(128);
                . . .
        }

private:
        AccessMethods<int, string>* pAm_;
}

////////////////////////////////////
#include "myclass.hpp’’
#include "am/btree.hpp’’
#include "am/rtree.hpp’’
#include "am/am.hpp’’
int main()
{
        AccessMethods<int, string>* pAm = new BTree<int, string>(...);
        MyClass test1(pAm);
        test1.myFoo();
        delete pAm;
        . . .
        pAm = new RTree<int, string>(...);
        MyClass test2(pAm);
        test2.myFoo();
        delete pAm;
        . . .
}
```

- You can make your modules more reuseable.

```
template<typename AmType>
void foo(AmType& am)
{
        am.insert(128, ‘‘Hello! AM-lib‘‘);
        am.getDataBy(128);
}

////////////////////////////////////
#include "am/btree.hpp’’
#include "am/rtree.hpp’’
int main()
{
```

```
                        BTree<int, string> btree;
                        foo(btree);
                        . . .
                        RTree<int, string> btree;
                        foo(btree);
                        . . .
                }
```

- You don’t need to worry about the size of data, cause AM-lib will use disk when data size comes very large.

```
                void foo(AmType& am)
                {
//initialize a 3 dimensions dynamic array
                        MulDimDynArray largeArray(100000000, 100000000, 1000000);
                        MulDimDynArray smallArray(10, 10, 10);
                        smallArray.append(largeArray);
                        . . .
                }
```

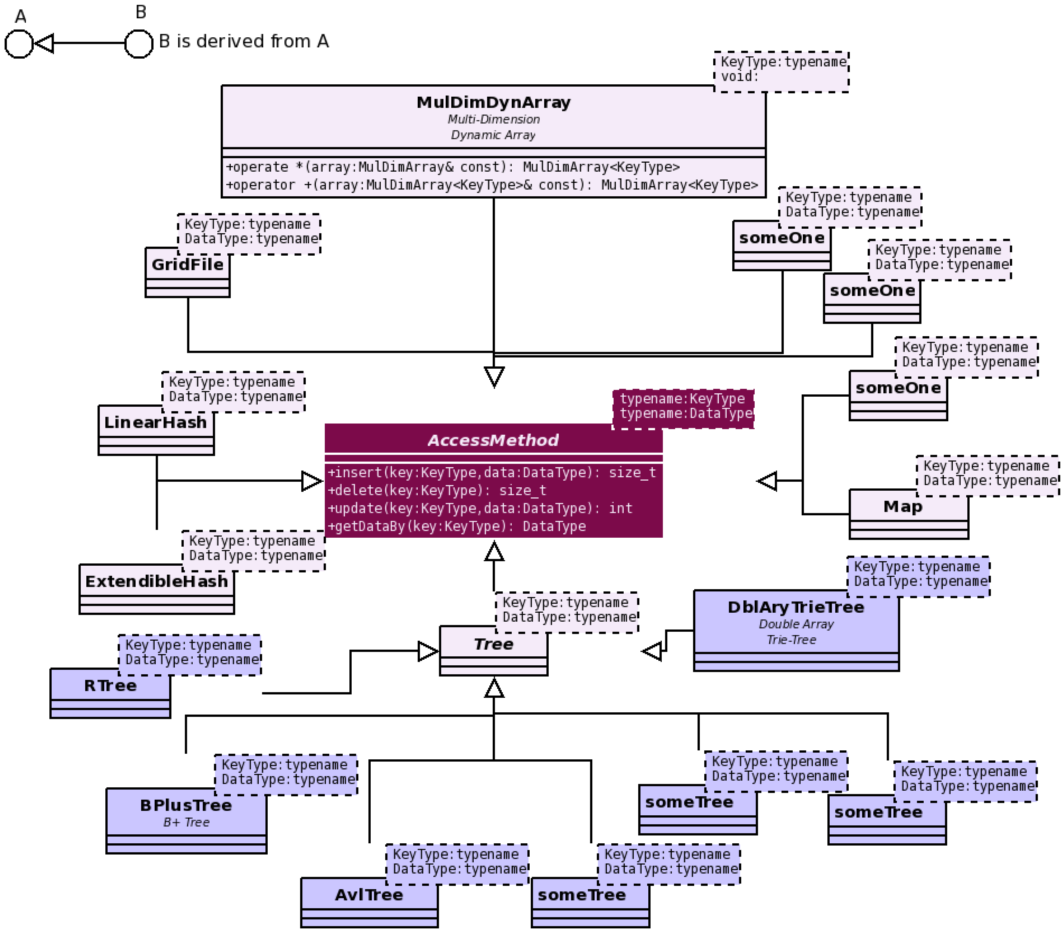

Figure 1: Initial design of AM-lib classes

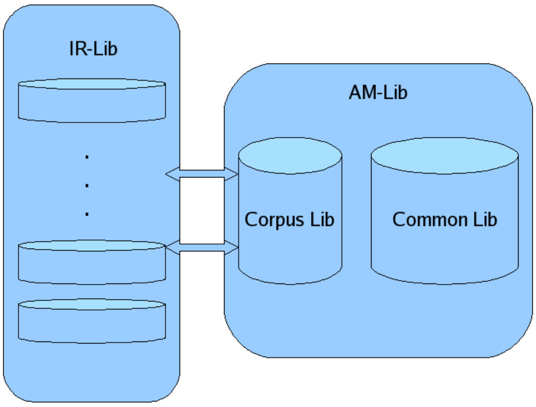

Figure 2: AM-Lib and IR-LIB

4 Corpus Management

Corpus management component of AM-Lib provides storage services for IR-Lib. It is necessary because each component of IR-Lib will read data from corpus and output the results to a generic structs, therefore refactoring this common part into a single library will improve the system's reusage.

Existing project as *SML* provides similiar components as DocumentBag,etc. We can base corpus management on SML. What's more, IR-Lib needs a powerful matrix component to store the middle temporary computation outputs and the ultimate results.

4.1 Design of Matrix Library

Matrix is one of the most important fundamental component that is required by all kinds of machine learning and information retrieval algorithms. Besides the general numeric matrix utilities, we plan to include the following characteristics:

- A general matrix that has both memory and file version.

  File version is important because of the large scale data to be dealed with.

- SVD Decomposition, QR Decomposition, LU Decomposition, Eigenvalue Decomposition.

  These utilities are extremly useful in machine learning and information retrieval.

- Hessian matrix

  Hessian matrix is useful in Bayesian inference and decision theory.

Since the matrix library is required to be implemented with a modern generic design, what's more, a matrix library that provides general numeric utilities is itself a large work, therefore, we should base our implementation on the existing library, or else, the matrix library would have been a large engineering work. Taking the consideration of the file version matrix, together with the generic design, it means that the library that we choose to base on should be totally hacked, or else both of these two requirements could not be satisfied. *uBlas* is the matrix library included by *Boost*, it could be one of the choice to base on, however, it still has the following limitations:

- *uBlas* is frequently changed with the *Boost* library, since our code would be integrated with the existing library, it means that each engineer that uses different version of

Boost perhaps could not get the matrix library worked, and also, whenever *Boost* upgraded its library(with a frequency of 2 months per time or so), we should change our matrix design to keep up with the new changes.

- According to *Ian*'s experience on *uBlas*, it still has some bugs, which leads to redesigning a basic matrix utility in *Clustering-framework*.

There exist two other generic matrix library—*Mtl4*[1], and *Glas* [2], which are also total generic design with a better architecture and code quality. Still, we have not decided which library would be chosen to be based on, all of these libraries have the same design idea and take time to study to be able to hack them.

As to the issue of file version of matrix library, we have got a new idea that utilizing the *mmap* system call to map the file to the process space, after this step, we could operate file the same as memory. Under 64bit mode, the size limit of the file to be mapped is large enough to satisfy all kinds of huge matrix data. Following is the sample code which is platform independent.

```
#ifndef WIN32
void* ptr = ::mmap(0, len, prot, MAP_SHARED, fd, 0);
#else
DWORD prot = 0;
HANDLE h;
void* ptr = NULL;
h = CreateFileMapping((HANDLE)_get_osfhandle(fd), 0, PAGE_READWRITE, 0, len, 0);
if (h != INVALID_HANDLE_VALUE) {
ptr = MapViewOfFile(dm_mappingobj, FILE_MAP_ALL_ACCESS, 0, 0, len);
}
#endif
```

This kind design could improve the code reuage, additionally,after the file is mapped into the process space, we can reduce the context between os kernel space and user process space, which leads to a better IO performance.

5 Components of IR-Lib

IR-Lib is a collection of algorithms in machine learning and information retrieval, together with the Corpus Management component of AM-Lib, it could provide a generic framework for search applications. Both machine learning and information retrieval have covered lots of fields, therefore, the main purpose of IR-Lib is to provide a scalable framework together with general algorithms, then in future, more advanced algorithms could be added easily.

The relationship between machine learning and information retrieval is very close and machine learning could be seen as the lower layer to provide methods for information retrieval's usage. Therefore IR-Lib could be composed of two layers. In addition, there exists some fields in information retrieval that has not adopted methods provided by machine learning, such as recommendation systems, preprocessing, etc. We will talk about all the components of IR-Lib one by one.

6 Machine learning Components

- Supervised Learning.
  Wisenut-classifier has implemented most of the basic supervised learning methods. We plan to replace the interface to *SML* with the interface to new corpus management component of AM-Lib, and then refactor it to the generic design.

[1] http://www.osl.iu.edu/research/mtl/mtl4/

[2] http://www.cs.kuleuven.be/˜ karlm/glas/

- Unsupervised Learning.
  Clustering-framework has already doen a good job of it. Therefore, the relavant job of this component is to make *Clustering-framework* suit for the whole framework of IR-Lib.

- Learning Complex Models.

  1. EM(Expectation—Maximization), which is also a basic learning approach in semi-supervised learning.
  2. Hidden Markov Models.
  3. Sampling method, including MCMC.
  4. Graphical Models, graphical models including following directions, each of which is under hot research, we are not sure whether it is possible to implement all of them, just try to do that.
     (a) Bayesian Network.
     (b) Markov Random Fields.
     (c) Conditional Random Fields.

- Dimensionality Reduction. We plan to implement PCA at first, more approaches could be done in future if possible.

In summary, machine learning are still under fast developing process, therefore only some basic directions would be included into this library, we hope a good design framework could be provided in order that more learning approaches could be included into this library easily in future.

7 Information Retrieval Components

- Text Pre-Processing
  Text pre-processing techniques are mature and have been implemented by existing projects, therefore we can refactor them from existing code.

  1. Stopword Removal
  2. Stemming
  3. TF-IDF
  4. Tokenization
  5. Feature Selection
  6. Duplicate Detection

- Language Models
  It is necessary to refactor and integrate Jinglei's work into the library.

- Topic Modeling
  Topic modelling is a hot research direction and lots of new approaches appear continuously. We only plan to provide some topic modelling methods including LSI, LDA and 4-level PAM. We hope more topic modeling approaches could be easily added to this library.

8 Access Methods Library Design

8.1 Summery of exiting library

As Figure 3 shows, YLIB is a great work which includes 5 parts as algorithm, container, database, data processor and network. And SML focuses on how to deal with corpus and categorize. SF1Lib is a new library focuses on manipulating database. Our work is giving these library a new face using generic programming. For so many modules and classes in these library, just a few of them relates with the concept access methods. And we conclude the basic functions of AM is QUID (querying, updating, insertion and deletion). Policy-based design in book 'Modern C++ Design: Generic Programming and Design Patterns Applied' gives us a very good clue to do this work.

8.2 Policy-based AM

Policy-based design treats every changeable parts as a policy and integerates all the policies in a host which is treated as an interface for application developers and coordinates all the policies to give the functions. What a user of policy-based designed library need to know is what kind of policy he wants to use and where's the host class. Thus, what are the changeable parts of AM-Lib? They are the place to store the data, main memory or disk, if data stored on disk, the type of cache used, and data type, key type, and the container with different algorithms. We can divide access methods and data structure into two parts. One is some kind of easy data structure without specific algorithms like vector, list etc. The other is some complex structure like tree and table with some certain algorithm. The design of these two parts should be different but their interface to the client should be the same. The simple data structure could be like follow.

```
//Library code
template
<
         class T,
         class StorageStrategy = MemStorage//if data size is very large,
                                           //we need to change it into DiskStorage
>
class Array;
///////////////////////////////////////////
template
<
         class T,
         class StorageStrategy = MemStorage
>
class List;
//////////////////////////////////////////
. . .
///////////The main class for these simple data structure//////////
template
<
         class T,
         template<class, class> class ConcreteDS,
         class SearchStrategy = BinarySearch,
         class SortStrategy = QuickSort,
         class StorageStrategy = MemStorage
>
class SimpleAccessMethod : public ConcreteDS<T, StorageStrategy>;
```

Some kind of complex DSs (data structures) are different. The data manipulated by these DSs are composited of key and data. They have their own algorithms for sorting and searching. And for some complex situations, the key could be multiple.

```
//Library code
template
<
         class KeyType,
        class DataType,
        class StorageStrategy = MemStorage//if data size is very large,
                                          //we need to change it into DiskStorage
>
class BPTree;
///////////////////////////////////////////
template
<
        class KeyType,
        class DataType,
        class StorageStrategy = MemStorage
>
class HashTable;
//////////////////////////////////////////
. . .
///////////The main class for these complex data structure//////////
template
<
        template<class> class KeyTypeList,
        class DataType,
        template<class, class, class> class ConcreteDS,
        class StorageStrategy = MemStorage
>
class ComplexAccessMethod : public ConcreteDS<KeyTypeList<ConcreteDS>, StorageStrategy>;
```

The final step is to combine SimpleAccessMethod and ComplexAccessMethod together to have a generic face for client to call. To be continued ...

8.3 Storage Manager

8.3.1 Memory Management

Most of the C++ programmers do not benefit from 'Garbage Collection' technique (GC). They are sick of deleting objects but have to do this. There are some C/C++ memory GC implementations, but they are complex and are not widely used. Although we have smart pointer which is based on 'Reference Counting', it is not always a good idea however:

- It's a fact that not all of the C++ programmers like smart pointers, and not all of the C++ programmers like the SAME smart pointer. You have to convert between normal pointers and smart pointers, or between one smart pointer and another smart pointer. Then things become complex and difficult to control.
- Having a risk of Circular Reference.
- Tracking down memory leaks is more difficult

Therefore it is recommended to introduce the *GCAllocator* included by *StdExt* when allocating small objects.

8.3.2 File Management

STXXL is an extremely high efficient file based container, where there exists a file block manager inside. It is necessary to extract the file manager to form a file based allocator, then it would be scalable to extend from a memory version data container to a file version with high performance.

8.4 Minimal Perfect Hashing

Minimal perfect hashing(MPH) could be divided into two categories–order-preserving and non order-preserving. Order-preserving is more useful in information retrieval, however, more research results have been got on non order-preserving solutions. Both of the MPH would be added into *iZENELib*. Like B-Tree component, MPH also provides an iterator for sequential access.

9 Appendix

Table 1: Implement schedule

	Miles-tone	**Start**	**Finish**	**In Charge**	**Description**	**Status**
1	Preparation for *iZENELib*	2008-11-01	2008-12-05	Yingfeng, Kevin	Study the generic design idea, refactor *YLib*,find solutions for important utilities including matrix, file block manager.	Finished
2	AM Interface Definition	2008-12-08	2008-12-12	Yingfeng, Peisheng, Kevin	Make sure the policy based AM interface	Finished
3	Encapsulation for basic AM methods	2008-11-01	2008-12-31	Kevin,	Encapsulate Linear hash table memory version, skip list memory version, etc.	Memory versions done
				Peisheng	Encapsulate sequencial DB, B-tree, skip list file version, etc.	In progress
				Liang	Encapsulate Priority Queue, etc.	In progress
				Vernkin	Encapsulate Perfect hash, etc.	In progress
4	Storage manager	2008-12-08	2008-12-24	Yingfeng	An extremely efficient memory allocator together with a file block allocator should be provided,the former could improve memory version of AM methods remarkably and the latter could support file based data structure with high scalability and high performance	Todo
5	Matrix library	2008-12-25	2009-01-14	Yingfeng	An efficient matrix library is provided	Todo
6	Basic IR Library	2009-01-14	2009-01-31	Yingfeng	Basic IR-Lib could be provided, including corpus management, basic supervised and unsupervised learning	Todo
7	Pre-Processing	2009-01-01	2009-01-15	Kevin	Text pre-processing component is encapsulated.	Todo
8	Complex machine learning	2009-02-01	2009-02-28	Yingfeng	Learning methods for complex models have been added.	Todo
9	Network Library	2009-03-01	2009-03-31	Yingfeng	Distributed computing component is added.	Todo

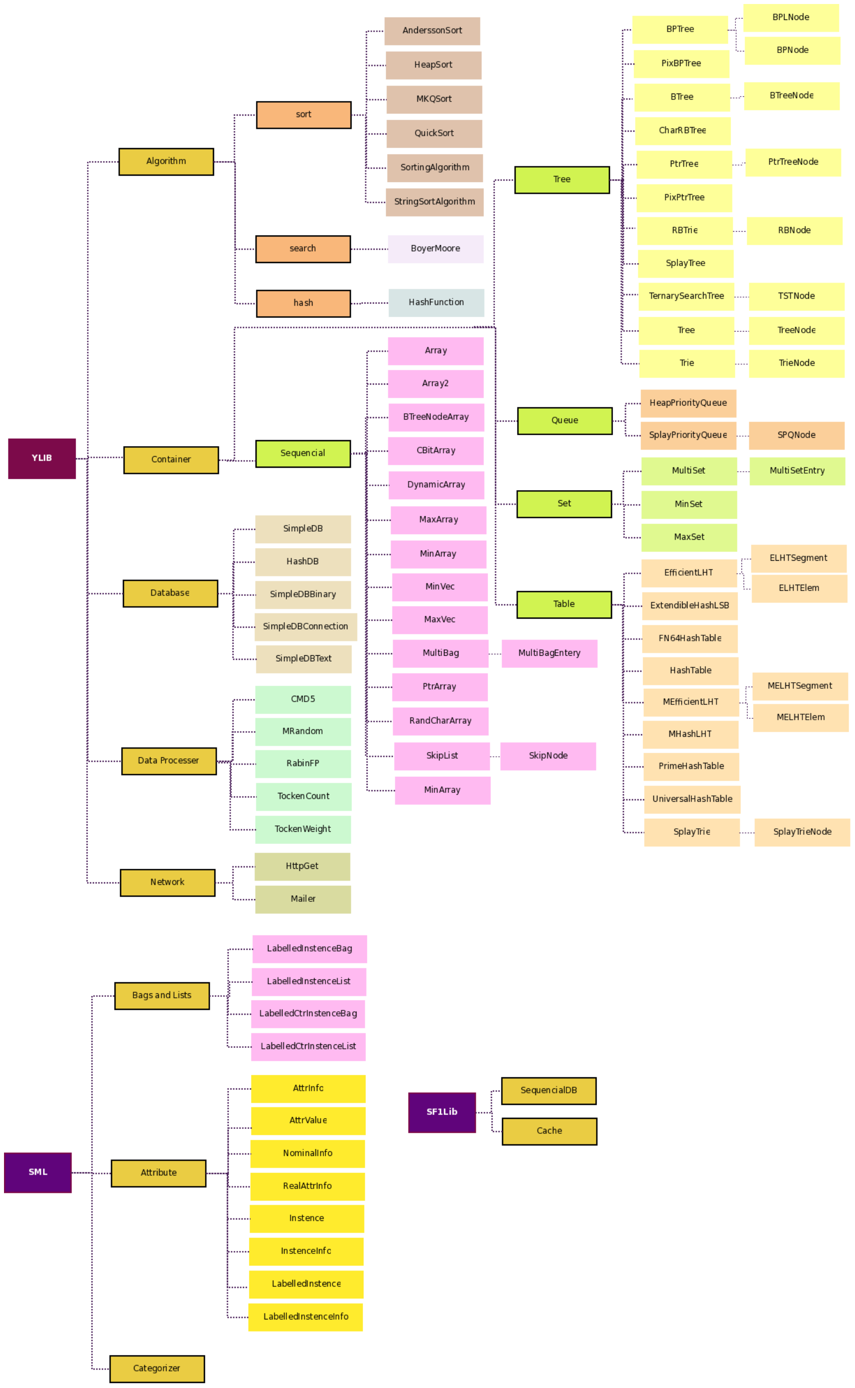

Figure 3: Summery of exiting library