Technical Report of iZENELib

Yingfeng Zhang, Kevin Hu, Peisheng Wang

December 29, 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 **2**

2 Design Goal **2**

3 Access Methods Library Design **2**
3.1 Requirements . . . 2
3.2 Summery of exiting library . . . 3
3.3 Policy-based AM . . . 3
3.4 Storage Manager . . . 8
3.4.1 Memory Management . . . 8
3.4.2 File Management . . . 8
3.5 Minimal Perfect Hashing . . . 10

4 Corpus Management **10**
4.1 Design of Matrix Library . . . 11
4.1.1 Boost.uBLAS . . . 11

5 Components of IR-Lib **12**

6 Machine learning Components **12**

7 Information Retrieval Components **12**

8 Utility components **13**
8.1 iZeneLib Log . . . 13

9 Appendix **14**

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.
2008-12-19	peisheng	Update KeyType, ValueType and DataType description.

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 Access Methods Library Design

3.1 Requirements

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);
        . . .
}
```

3.2 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.

3.3 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.

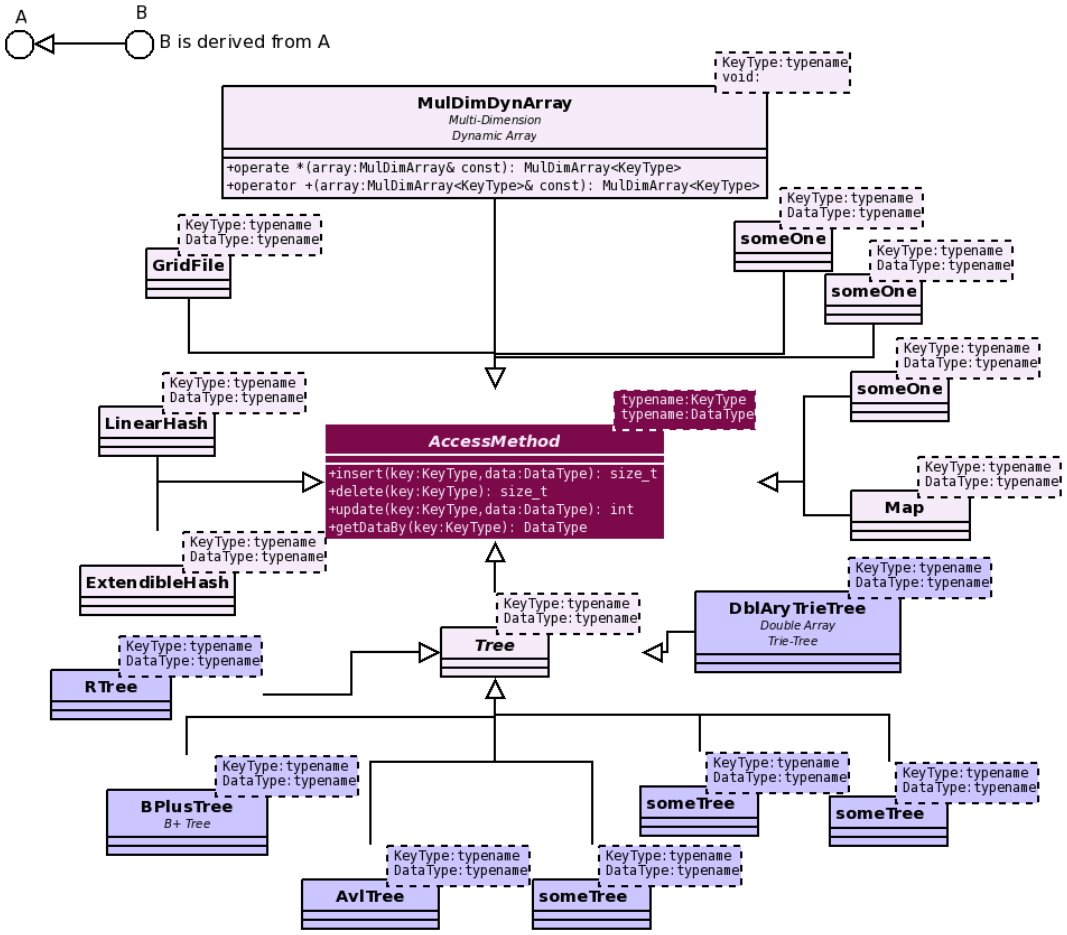

Figure 1: Initial design of AM-lib classes

```
template<typename KeyType, typename ValueType,
    typename LockType=NullLock, typename Alloc=std::allocator<DataType<KeyType,ValueType> > >
class AccessMethod
{
public:
virtual bool insert(const KeyType& key, const ValueType& value);

virtual bool insert(const DataType<KeyType,ValueType>& data) = 0;

virtual bool update(const KeyType& key, const ValueType& value);

virtual bool update(const DataType<KeyType,ValueType>& data) = 0;

virtual ValueType* find(const KeyType& key) = 0;

virtual bool del(const KeyType& key) = 0;
};

template<typename KeyType, typename LockType=NullLock,typename Alloc=std::allocator<DataType<KeyType> > >
class UnaryAccessMethod
{
public:
virtual bool insert(const KeyType& key);

virtual bool insert(const DataType<KeyType>& data) = 0;

virtual bool update(const KeyType& key);
```

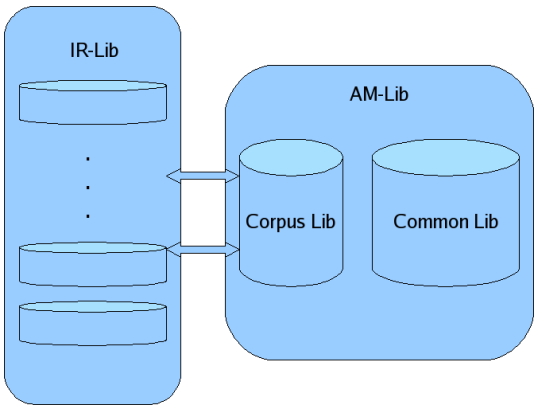

Figure 2: AM-Lib and IR-LIB

```
virtual bool update(const DataType<KeyType>& data) = 0;

virtual KeyType* find(const KeyType& key) = 0;

virtual bool del(const KeyType& key) = 0;
};
```

All binary components of *AM-LIB* should implement the interface of AccessMethod while all unary components should implement the interface of , where LockType is the thread policy which allows for different threading model to be applied, Alloc is the memory policy, which we could choose to store elements in memory or file. The elements of the *AM-LIB* are **DataType**, which contains **KeyType** and **ValueType**, when **ValueType** is **NullType**(defined below), it is unary.

DataType provide **get_key()**, **get_value()**, **serialize()**, and **compare()** method.

```
struct NullType{
};

//When ValueType is NullType, it is equivatent to unary DataType.
template<typename KeyType, typename ValueType=NullType>
class DataType
{
public:
    DataType(){ }
    DataType(const KeyType& key, const ValueType& value)
    :key(key), value(value)
    {
    }

    int compare(const DataType& other) const
    {
        return _compare(other, static_cast<boost::is_arithmetic<KeyType>*>(0));
    }

    template<class Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        ar & key;
        ar & value;
    }
    const KeyType& get_key() const {return key;}

    const ValueType& get_value() const {return value;}
private:
    int _compare(const DataType& other, const boost::mpl::true_*) const
    {
```

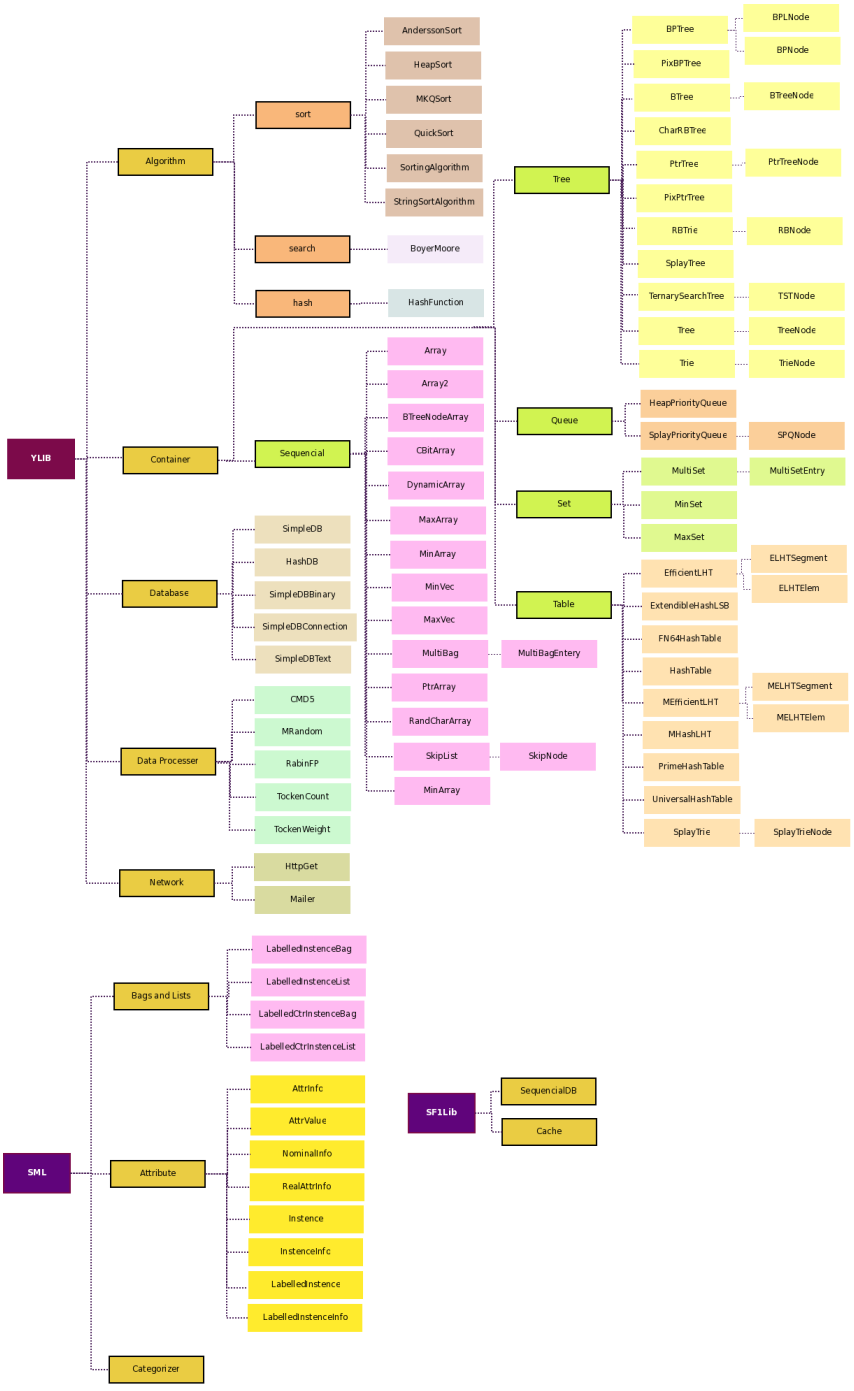

Figure 3: Summery of exiting library

```
    return key-other.key;
}
int _compare(const DataType& other, const boost::mpl::false_*) const
{
    BOOST_CONCEPT_ASSERT((KeyTypeConcept<KeyType>));
    return key.compare(other.key);
```

```
    }

public:
    KeyType key;
    ValueType value;
};

template<typename KeyType>
class DataType<KeyType, NullType>
{
public:
    DataType(){}
    DataType(const KeyType& key)
    :key(key)
    {
    }
    DataType(const KeyType& key, const NullType&)
    :key(key)
    {}

    int compare(const DataType& other) const
    {
        return _compare(other, static_cast<boost::is_arithmetic<KeyType>*>(0));
    }

    template<class Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        ar & key;
    }

    const KeyType& get_key() const {return key;}


private:
    int _compare(const DataType& other, const boost::mpl::true_*) const
    {
        return key-other.key;
    }

    int _compare(const DataType& other, const boost::mpl::false_*) const
    {
        BOOST_CONCEPT_ASSERT((KeyTypeConcept<KeyType>));
        return key.compare(other.key);
    }

public:
    KeyType key;
};
```

As for meta type **keyType** like int, float that don't have **compare()** method, a default CompareFunctor is also provided.

```
template<class KeyType>
class CompareFunctor:public binary_function<KeyType, KeyType, int>

{
public:
    int operator()(const KeyType& key1, const KeyType& key2) const
    {
        return _compare(key1, key2, static_cast<boost::is_arithmetic<KeyType>*>(0));
    }
private:
    int _compare(const KeyType& key1, const KeyType& key2,  const boost::mpl::true_*) const
    {
        return key1 - key2;
    }

    int _compare(const KeyType& key1, const KeyType& key2,  const boost::mpl::false_*) const
    {
        BOOST_CONCEPT_ASSERT((KeyTypeConcept<KeyType>));
        return key1.compare(key2);
    }
};
```

All components of *AM-LIB* share the same definition of element—DataType.

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>;
```

3.4 Storage Manager

3.4.1 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. The detailed design and usage of *GCAllocator* could be got from another document—*GCAllocator.pdf*, written by *Xushiwei*—author of *GCAllocator*. It has been improved and some bugs have been fixed, locating at *izenelib/include/3rdparty/boost/memory*.

3.4.2 File Management

Since there might be lots of data structures that are required to provide a file based version to make data persistent, it is reasonable to provide a good file management mechanism to accelerate the development of file based version. *iZENELib* has provided two kinds of utilities to satisfy this requirement.

BlockManager *STXXL* is an extremely high efficient file based container, where there exists a file block manager inside, together with an asynchronous I/O layer(AIO).The purpose of the AIO layer is to provide a unified approach to asynchronous I/O. The layer hides details of native asynchronous I/O interfaces of an operating system and has the following advantages:

1. To issue read and write requests without having to wait for them to be completed.
2. To wait for the completion of a subset of issued I/O requests.
3. To wait for the completion of at least one request from a subset of issued I/O requests.
4. To poll the completion status of any I/O request.
5. To assign a callback function to an I/O request which is called upon I/O completion (asynchronous notification of completion status), with the ability to co-relate callback events with the issued I/O requests.

STXXL has a high performance on disk I/O, its I/O counterpart has been extracted independently to the *BlockManager* in *iZENELib*.However, it has the limitation of being able to process data with fixed length only. *BlockManager* is just a file block manager, with a *LRU* cache policy inside and an AIO layer, if it was adopted, still lots of work are needed, such as, how to manage data location,how to design cache,etc, however, these works are relevant to different application, which can not be easily abstracted to a common utility.

File Based Allocator Another policy of file management is to provide a file based allocator, having the same interface as such memory allocators as *std::allocator*, therefore, if an application is designed based on allocators, the according file based version is very easy to implement–just replace the original temlate parameter having value of *std::allocator* with the file based allocator. The only available mechanism of implementing such an allocator is to recur to a function provided by operating system—file mapping.Both *UNIX* like operating system and *Windows* have provided such an ability, with *mmap* for the former, *OpenFileMapping* for the latter. Under 32bit environment, there exists a limitation that only 2G bytes could be mapped into the virtual memory address at one time, therefore, for larger file, it is inconvenient to manage larger file. Under 64bit environment, such a limitation has been trivial because the virtual memory address is large enough. We have two choices for such kind of file based allocator—the one provided by *Boost* and the one provided by *iZENELib*. Take the former as the example:

```
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/managed_mapped_file.hpp>
#include <boost/interprocess/allocators/allocator.hpp>

using namespace boost::interprocess;
typedef list<int, allocator<int, managed_mapped_file::segment_manager> >  MyList;
int main ()
{
   const char *FileName       = "file_mapping";
   const std::size_t FileSize = 10000;
   std::remove(FileName);
   managed_mapped_file mfile_memory(open_or_create, FileName, FileSize);
   MyList * mylist = mfile_memory.construct<MyList>("MyList")(mfile_memory.get_segment_manager());

   //Obtain handle, that identifies the list in the buffer
   managed_mapped_file::handle_t list_handle = mfile_memory.get_handle_from_address(mylist);
   //Fill list until there is no more room in the file
   try{
      while(1) {
         mylist->insert(mylist->begin(), 0);
      }
   }
```

```
    catch(const bad_alloc &){
       //mapped file is full
    }
    //Let's obtain the size of the list
   std::size_t old_size = mylist->size();
    //To make the list bigger, let's increase the mapped file
    //in FileSize bytes more.
    mfile_memory.grow(FileName,FileSize);
    //If mapping address has changed, the old pointer is invalid,
    //so use previously obtained handle to find the new pointer.
    mylist = static_cast<MyList *>(mfile_memory.get_address_from_handle(list_handle));
    //Fill list until there is no more room in the file
    try{
       while(1) {
          mylist->insert(mylist->begin(), 0);
       }
    }
    catch(const bad_alloc &){
       //mapped file is full
    }
    //Let's obtain the new size of the list
    std::size_t new_size = mylist->size();
    assert(new_size > old_size);
    //Destroy list
    std::cout<<"old size "<<old_size<<std::endl;
    mfile_memory.destroy_ptr(mylist);
    return 0;
}
```

As shown above, file mapper in *Boost* is not so convenient because the length of the file has to be sure at first, and if the file space has been exhaust, the programmer has to increase the file mapping size manually. The essential design aim of the utility provided by *Boost* is to satisfy the inter process communication, therefore, the designer has not considered much about how to apply file mapping to storage design, that is why *iZENELib* still provides another implementation.With this implementation, the file could grow its size automatically, and a file space garbage collection is also provided. It is extremely easy to use, for example:

```
template<class T>
class vector : public std::vector<T, izenelib::am::allocator<T> >{};
```

With the above codes, we then have a file based vector, which has the same usage as *std::vector*.What's more, according to the benchmark testing, the file based container can perform even faster then its according memory version, following is the bench result between file mapping and *STL* containers given a data set with 1000000 items:

	Insert	Sequential Read	Random Read	Delete
filemapper	1710ms	1310ms	1870ms	780ms
std containerr	1660ms	1300ms	1910ms	790ms

iZENELib has also provided the *new* and *delete* operation to construct and destroy the object within the file space.

3.5 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.

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.

4.1.1 Boost.uBLAS

Boost.uBLAS provides templated C++ classes for dense, unit and sparse vectors, dense, identity, triangular, banded, symmetric, hermitian and sparse matrices. The storage design of *uBLAS* is based on *unbounded_array*, which adopts *std::allocator* to allocate and deallocate the memory of its internal elements, it is therefore means that although *uBLAS* provides an implementation of memory version only, we could replace *std::allocator* with our above referred persistent allocator to provide a file based matrix library very easily, for example:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <am/filemapper/persist.h>

int main () {
    using namespace boost::numeric::ublas;
    using namespace izenelib::am;
    //we assign a file to be used here
    map_file root("matrix.map",1, create_new|auto_grow, 1);
    compressed_matrix<double,row_major, 0,
            unbounded_array<std::size_t, izenelib::am::allocator<std::size_t> >,
            unbounded_array<double, izenelib::am::allocator<double> > > m (300, 300, 300 * 300);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

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.
- 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 Utility components

8.1 iZeneLib Log

Part of making them easier to develop/maintain is to do logging. Logging allows you to later see what happened in your application. It can be a great help when debugging and/or testing it. The great thing about logging is that you can use it on systems in production and/or in use - if an error occurs, by examining the log, you can get a picture of where the problem is. Good logging is mandatory in support projects, you simply can't live without it. Used properly, logging is a very powerful tool. Besides aiding debugging/ testing, it can also show you how your application is used (which modules, etc.), how time-consuming certain parts of your program are, how much bandwidth your application consumes, etc. - it's up to you how much information you log, and where.Here, we describe what this log can do and how to use the izenelib log. This log is based on boost logging. So, you can use it coupled with boost logging. Let's see an example as follow.

18:59.24 [dbg] this is so cool
18:59.24 [dbg] this is so cool again
18:59.24 [app] hello, world
18:59.24 [app] good to be back ;)
18:59.24 [err] it's an error here

Level There're three levels, debug/application/error infomation. Their formats are just like above. You can use some maroc like follow to output three different infomation. If you want to disable any kind of log informatin, just define some switch like DBG_DISABLE/ APP_DISABLE/ ERR_DISABLE.

LDBG_≪"Output debug information!";
LERR_≪"Output error information!";
LAPP_≪"Output application information!";

Tags The defualt tags for three different information is just like [dbg]/[app]/[err]. If you don't like them, you can define them. Respectively, you can define DBG_TAG/ APP_TAG/ ERR_TAG to specify the tags.

Destination As defualt, all the three kind information will be ouputed into three files, "'dbg.txt"/ "'app.txt"/ "'err.txt". The file names are defined by DBG_LOG_NAME/ APP_LOG_NAME/ ERR_LOG_NAME. Of course, you can output all the information into one file. All of thoes information will not be printed onto console. If you want to do it, you just need to define some switchs as DBG2CONSOLE/ APP2CONSOLE/ ERR2CONSOLE.

Conditional log Sometime, you want to log according some condition while the program is running. You can do it like :

IF_DLOG(i%10==0)≪"i mod 10 is 0!";

It's outputed as debug information. You can use other kind of log with *IF_ALOG()/ IF_ELOG()*.

Initialize log This is a kind of globle log within one program. You just need to initialize it in your *main()* function like this: ***USING_IZENE_LOG();***. Every time when you need to use log, don't forget to include the head file "util/log.h".

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.	Btree and SDB based on izenelib are finished and their efficiecy doen't decline.
				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	Finished
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