Technical Report of iZENELib

Yingfeng Zhang, Kevin Hu, Peisheng Wang

March 18, 2009

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.
2008-12-19	peisheng	Update KeyType, ValueType and DataType description.
2009-02-27	Kevin	Cache-concious hash table, B-trie and dynamic perfect hash
2009-03-18	Kevin	An external sort: AlphaSort.

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.

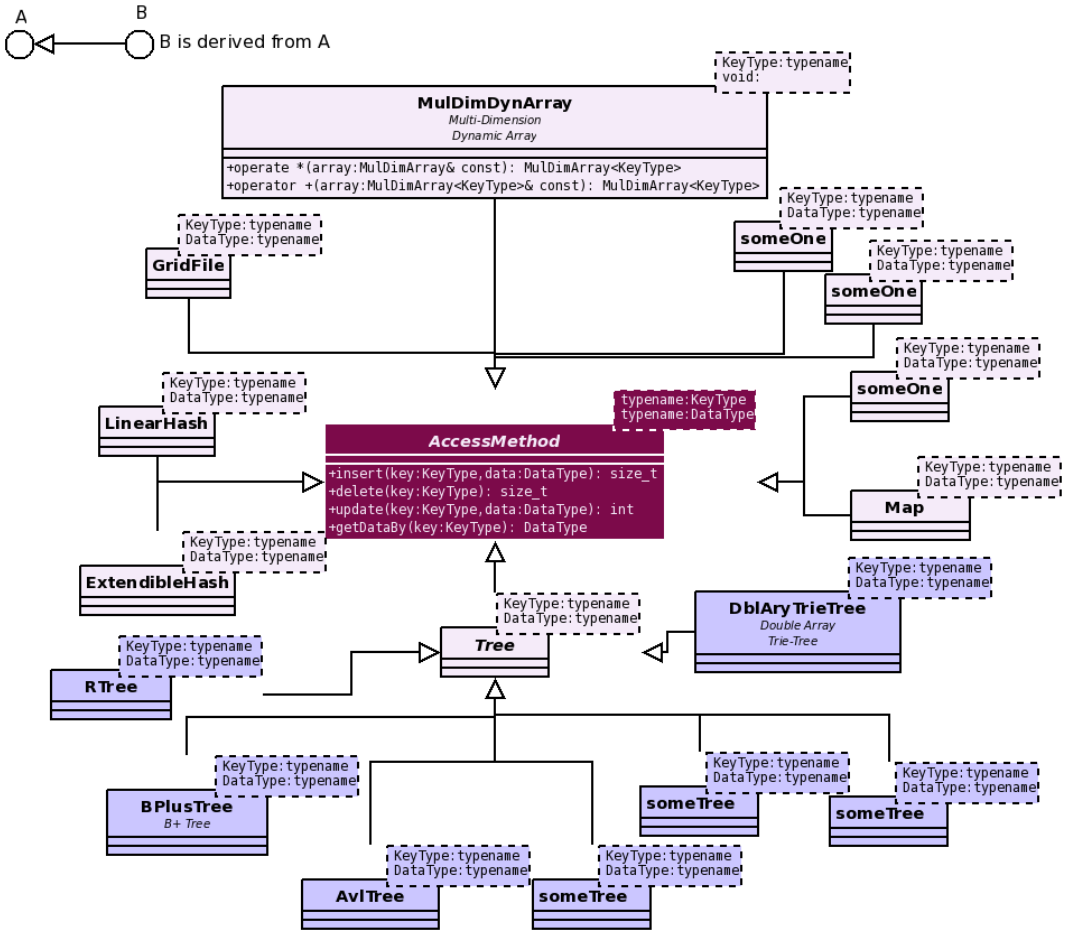


Figure 1: Initial design of AM-lib classes

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.

```
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;
```

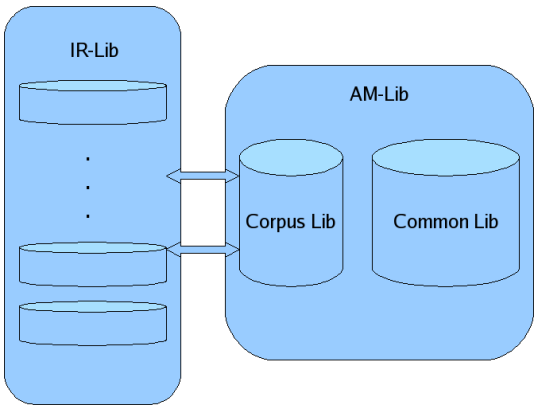


Figure 2: AM-Lib and IR-LIB

```
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);

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)
    {
```

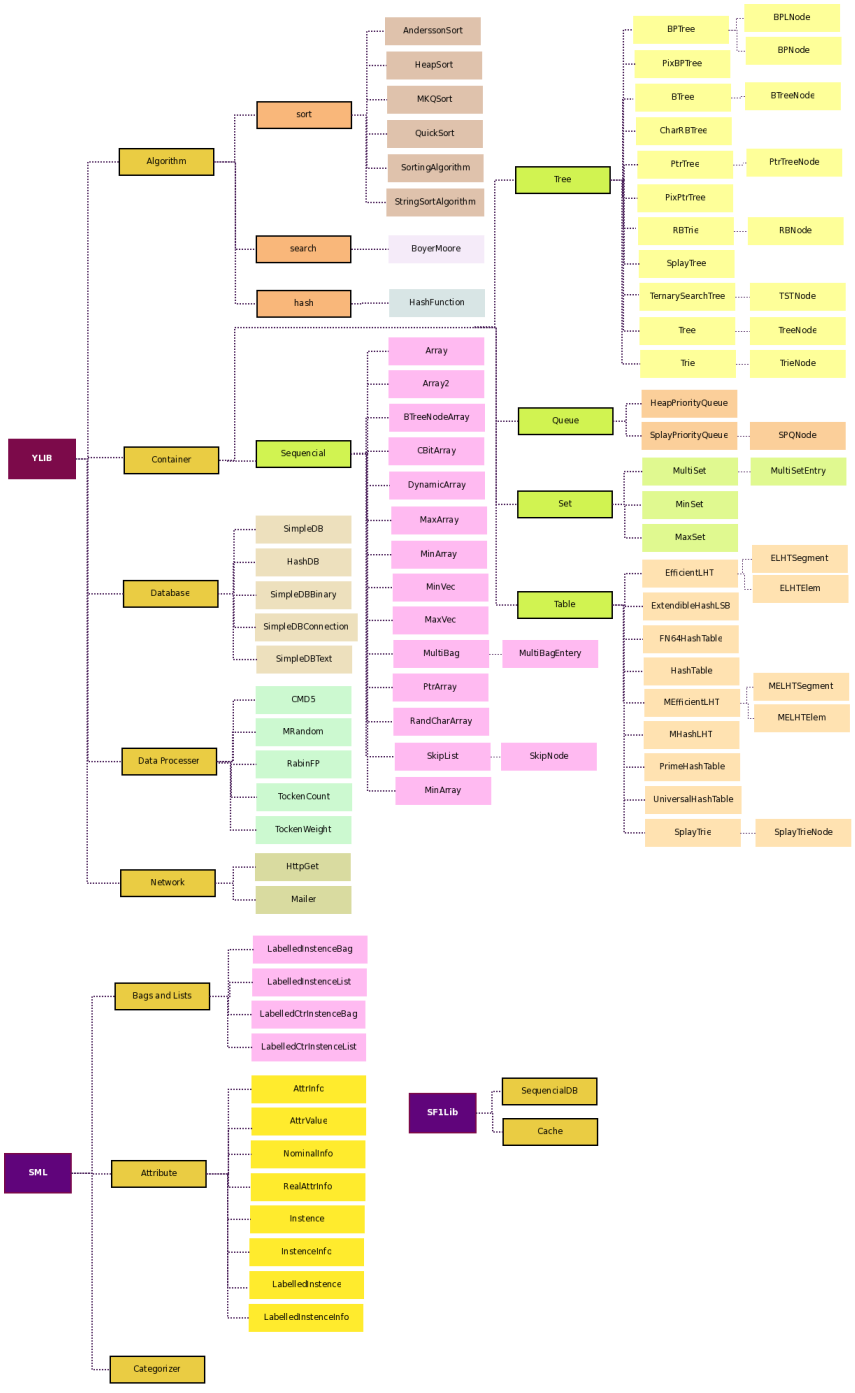


Figure 3: Summery of exiting library

```
}

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
    {
        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.

3.6 Dynamic Perfect Hashing

3.6.1 Perfect hashing

Although hashing is most often used for its excellent expected performance, hashing can be used to obtain excellent worst-case performance when the set of keys is static: once the keys are stored in the table, the set of keys never changes. Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM. We call a hashing technique perfect hashing if the worst-case number of memory accesses required to perform a search is O(1). The basic idea to create a perfect hashing scheme is simple. We use a two-level hashing scheme with universal hashing at each level. Figure 4 illustrates the approach.

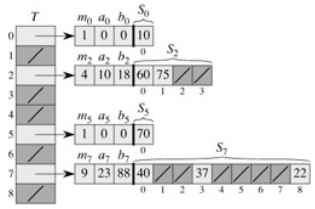


Figure 4: Perfect hash table

As Figure 4 show, it uses perfect hashing to store the set K = 10, 22, 37, 40, 60, 70, 75. The outer hash function is $h(k) = ((ak + b) mod p)\ mod\ m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T . A secondary hash table S_j stores all keys hashing to slot j . The size of hash table S_j is m_j , and the associated hash function is $h_j(k) = ((a_j k + b_j)\ mod\ p)\ mod\ m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

The first level is essentially the same as for hashing with chaining: the n keys are hashed into m slots using a hash function h carefully selected from a family of universal hash functions. Instead of making a list of the keys hashing to slot j, however, we use a small secondary hash table S_j with an associated hash function h_j. By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level. In order to guarantee that there are no collisions at the secondary level, however, we will need to let the size m_j of hash table S_j be the square of the number nj of keys hashing to slot j. While having such a quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirements to be excessive, we shall show that by choosing the first level hash function well, the expected total amount of space used is still $O(n)$.

3.6.2 Dynamic perfect hashing

This work is based on Martins paper Dynamic Perfect Hashing: Upper and Lower Bonds in 1990. During indexing, when conflict happens, it partially re-hashes data in this slot with

some random chosen hash function until find a hash function that no conflicts happen in this slot. Martin proved that at least half of the hash functions in random hash function set can do it. Random hash function means choosing parameters a, b, p. When the hash table is full (defined by Martin), it needs rehash all the table. First, the first level hash function should be chosen randomly again until some conditions(Martin has the details) meet. So, the parameter used for expanding hash table is credential. It can save the frequency of rehashing. We choose to double the current amount of slots.

Compared to Cache-conscious hash table, the insertion running time is almost the same if the perfect hash table initially large enough. In term of querying runtime, perfect hash table is almost 3 times faster than cache-conscious hash table. (I use the same 1,000,000 random integers respectively).

3.7 Cache-Conscious Collision Resolution String Hash Table

In-memory hash tables provide fast access to large numbers of strings, with less space overhead than sorted structures such as tries and binary trees. If chains are used for collision resolution, hash tables scale well, particularly if the pattern of access to the stored strings is skew. However, typical implementations of string hash tables, with lists of nodes, are not cache-efficient.

With the cost of a memory access in a current computer being some hundreds of CPU cycles, each cache miss potentially imposes a significant performance penalty. A cache-conscious algorithm has high locality of memory accesses, thereby exploiting system cache and making its behavior more predictable. There are two ways in which a program can be made more cache-conscious: by improving its temporal locality, where the program fetches the same pieces of memory multiple times; and by improving its spatial locality, where the memory accesses are to nearby addresses. Chains, although simple to implement, are known for their inefficient use of cache. As nodes are stored in random locations in memory, and the input sequence of hash table accesses is unpredictable, neither temporal nor spatial locality are high. Similar problems apply to all linked structures and randomly-accessed structures, including binary trees, skiplists, and large arrays accessed by binary search.

This work is based on Nikolas Askitis and Justin Zobel's work. Every node access in a standard-chain hash table incurs two pointer traversals, one to reach the node and one to reach the string. As these are likely to be randomly located in memory, each access is likely to incur a cache miss. In this section we explain our proposals for eliminating these accesses. We assume that strings are sequences of 8-bit bytes, that a character such as null is available as a terminator, and a 32-bit CPU and memory address architecture. We propose a novel alternative to eliminate the chain altogether, and store the strings in a contiguous array. Prefetching schemes are highly effective with array-based structures, so this array hash table (shown, with the alternatives, in Figure 5) should maximize spatial access locality, providing a cache-conscious alternative to standard and compact chaining. Each array can be seen as a resizable bucket. The cost of access is a single pointer traversal, to fetch a bucket, which is then processed linearly. The experiment shows it's much faster than dynamic hash table which is one of the fastest linear hash proposed by Parson.

3.8 B-trie for disk-based string management

3.8.1 Basic trie

A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated

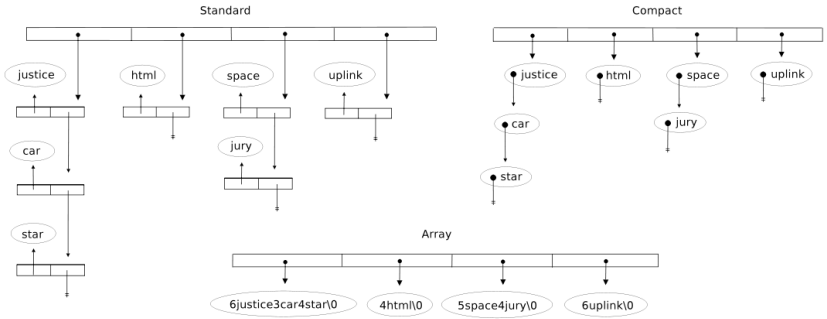


Figure 5: The standard-chain (left), compact-chain (right) and array (below) hash tables

with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest. So, every inner node of trie has to maintain an alphabet of corresponding language. For Chinese, there are about three thousands characters usually used. That is to say every node will be thousands bytes, which is really bad for memory performance and running time. And the large memory usage become the main problem to trie.

3.8.2 B-trie

The B-trie is an unbalanced multi-way disk-based trie structure, designed to sort and cluster strings that share common prefixes. It saves memory by put part of strings together into a bucket. The main components of our B-trie are as follows:

Trie nodes A trie node is an array of pointers, one pointer per character. Say, the ASCII table, there're 128 pointers in total. The prefix of a string is consumed by these nodes. Pointers in node point to other nodes or buckets.

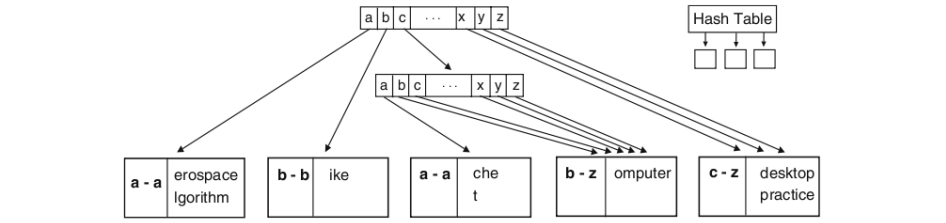
Buckets A leaf node of trie. A container stores strings after consumed by trie nodes. There're two kinds of bucket, pure bucket and hybrid bucket. Pure bucket stores strings which share the same first character. Hybrid bucket stores strings starting with different first characters.

Hash table Not every string inserted into trie is stored in it. Some relatively short strings are consumed by nodes and don't make it to leaf. Those strings will be stored in hash table.

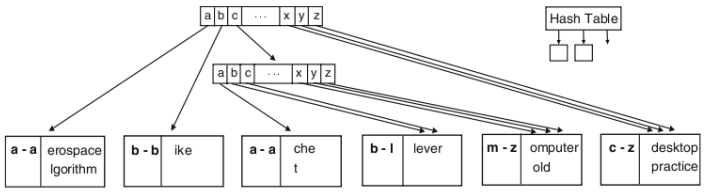
It borrows the design of the burst trie to maintain a space-efficient trie, by storing strings within buckets that are structurally similar to those described for the B+-tree: fixed-sized disk blocks represented as arrays. Once a bucket becomes full, a splitting strategy is required that, in contrast to bursting, throttles the number of buckets and trie nodes created. The concept of a B-trie has been suggested by Szpan kowski. However, information about the data structure, such as algorithms to insert, delete, search, and how to split nodes efficiently on disk, is scarce.

3.8.3 Bucket splitting

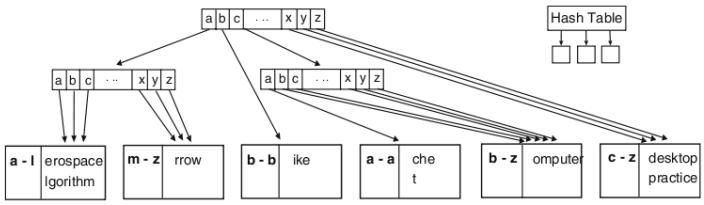
The most important operation of B-tire is splitting. We propose that buckets undergo a new splitting procedure called a B-trie split. When a bucket is split, a character is first selected



The words "cat","algorithm","computer", "practice", "cache", "bike", "desktop" and "aerospace" were inserted into the B-trie, creating three pure buckets (first three from the left) along with two hybrids. The hash table stores strings that are consumed. "c" for example, would be consumed by the root trie and "a", would be consumed by the first pure bucket



The strings "cold" and "clever" were inserted into the B-trie in Fig. 3. The second hybrid bucket (from the right of Fig. 3) split, creating two new hybrids



The word 'arrow" was inserted into Fig. 3. The left-most pure bucket split into two hybrids and a new parent trie

Figure 6: B-trie insertion progress

as a split-point and the strings are then distributed according to their leading character. That is, strings with a leading character smaller than or equal to the split-point remain in the original bucket, while others are moved into the new bucket. In this approach, the set of strings in each bucket is divided on the basis of the first character that follows the trie path that leads to the bucket. (Each node in the path consumes one character.)

When all the strings in a bucket have the same first character, this character can be removed from each string; subsequent splitting of this bucket will force the creation of a new parent trie. We label these buckets as pure. When the set of strings in a bucket have distinct first characters, several paths in the parent trie lead to it. We label these buckets as hybrid; subsequent splitting of this bucket will not create a new parent trie. Fig 6 illustrate examples of this splitting procedure.

In either case (hybrid or pure), each bucket is a cluster of strings with a shared prefix, a property with clear advantages for tasks such as range search. In addition, the B-trie offers other advantages. One is that the cost of traversing a chain of trie nodes can be, in comparison to the traversal of internal B-tree nodes, significantly lower; identification of a bucket involves no more than following a few pointers. Another is that short strings which are the commonest strings in applications such as vocabulary management are likely to be found without accessing a bucket and can be conveniently managed in memory. This splitting

process is, however, a major contribution, as it solves the problem of efficiently maintaining a trie structure on disk for common string processing tasks. A potential drawback compared to a B+-tree, is that splitting a bucket cannot guarantee that the two new buckets are equally loaded. In most cases, the load is likely to be approximately equal. Another drawback is the applicability of bulk-loading. To bulk-load a data structure implies populating leaf nodes without consulting an index. This is accomplished by using sorted data; the index is constructed independently as the leaf nodes are sequentially populated. Bulk-loading is an efficient way of constructing B+ -trees. However, the B-trie cannot be efficiently bulk-loaded because its index which can consume strings is not independent from the data stored in buckets.

3.8.4 Implementation

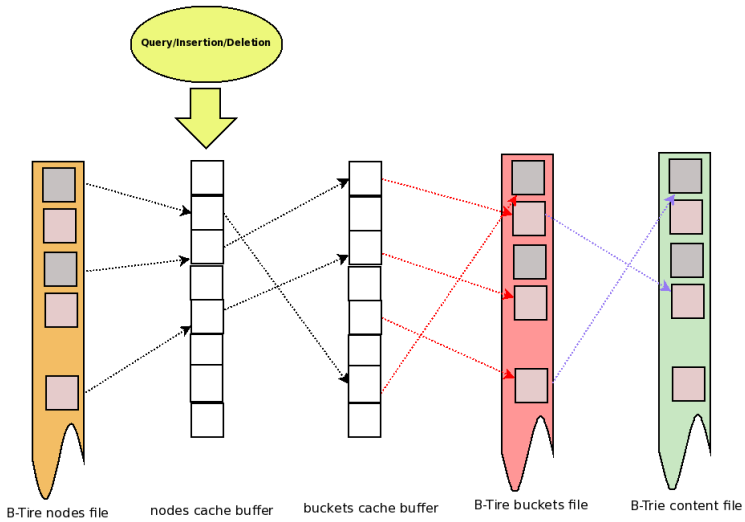


Figure 7: B-trie implementation architecture

For full usage of memory in B-trie, we adopt caches for fast memory access. Fig 7 shows the architecture of our implementation of B-trie. Except the data of tire node and bucket on disk, there're two caches for them respectively. So, the operation like insertion, updating etc. firstly will be executed in memory. If the required data doesn't exist in cache, then, it will be loaded from disk. The cache strategy is that latest and most frequently visited ones stay, and older and rarer visited ones out. We generate 1,000,000 random strings with length limits, and used these strings to test B-trie and B-tree. The tables below have detail.

Table 1: B-Trie performance with different bucket size to $1,000,000$ random strings

Bucket size(byte)	Str max len	Insertion(s)	Memory(MB)	Disk(MB)	Self-Query(s)
2000	20	4.6	320	30.3	3.35
	200	7.96	519	202.4	4.66
8196	20	4.53	208	33.3	3.36
	200	5.76	381	141.1	4.61

Table above shows the experiment results by testing 1,000,000 random string with max length 20 and 200. The content of this table includes inserting running time, memory and disk consumption, and accumulative querying time for searching for those 1,000,000

strings, respectively, with different bucket size and max string length. The bucket size of this algorithm is credential.

Table 2: B-Tree performance to $1,000,000$ random strings

Str max len	Insertion(s)	Memory(MB)	Disk(MB)	Self-Query(s)
20	6.78	-	-	-
200	10.02	-	300	-

According this above table of B-Tree's performance, B-trie has advantages comparing B-tree in some aspects while dealing with something like dictionary looking-up.

3.9 An external sort: AlphaSort

This work is based on Chris Nyberg's work "AlphaSort: A Cache-Sensitive Parallel External Sort" in 1995. A new sort algorithm, called AlphaSort, demonstrates that commodity processors and disks can handle commercial batch workloads. Using commodity processors, memory, and arrays of SCSI disks, AlphaSort runs the industry-standard sort benchmark in seven seconds. This beats the best published record on a 32-CPU 32-disk Hypercube by 8:1. On another benchmark, AlphaSort sorted more than a gigabyte in a minute.

AlphaSort is a cache-sensitive memory-intensive sort algorithm. We argue that modern architectures require algorithm designers to re-examine their use of the memory hierarchy. AlphaSort uses clustered data structures to get good cache locality. It uses file striping to get high disk bandwidth. It uses QuickSort to generate runs and uses replacement-selection to merge the runs. It uses shared memory multiprocessors to break the sort into subsort chores. Figure 8 shows the work flow of this algorithm.

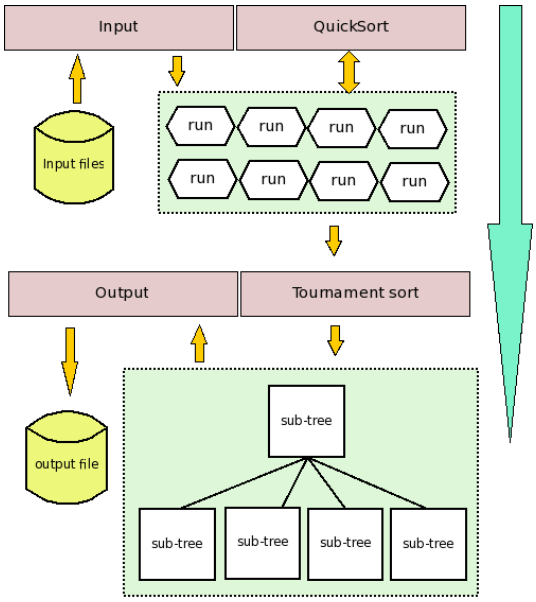


Figure 8: Work flow of AlphaSort

3.9.1 Input records

The records to be sorted have to include key field and content field. Content field follows the key field directly. No matter how long is the key field, we only use 4 bytes (user can

define how many bytes) of the key as pre-key to compare each other. If two records have same pre-key, we use next 4 bytes to compare. Consequently, we may use part of record content to compare when the key is the same. That makes sense. In that way, we would not worry about the key type users may give. And the content of the records may be very long. It costs a lot to copy records while sorting. So, we only load pre-key into memory, and add record's file handler and address together as representative of a record.

3.9.2 Cache-sensitive quick-sort

We group inputs into small groups which can be hold in CPU cache. We call that small group a run. Records in one group can be sorted without cache miss. That means hundreds times promotion of run-time compared to in-memory-sort. Because of the limited memory, some sorted runs must be stored in some temporary file for the next step: merge to use.

3.9.3 Cache-sensitive tournament tree sort

After every run is sorted, a tournament tree should be built to merge runs. Tournament tree is kind of binary tree. parent node is the smaller/bigger one of child nodes. The root node is the smallest/biggest one in a tree. Initially, the first records of every runs are used to build the tree. So, the root is the smallest/biggest one of the all the records. And next record would be added into tree in place of the root record should come from the same run as the current root record does. The output node of this tree-sort is put into a buffer while output procedure is reading from this buffer and writing them into output file in disk.

In order to use cache efficiently, we split the binary tree into blocks whose size fit cache size. And one block represent a sub-tree. Within a block, we use array to store this sub-tree. So, there is no cache miss of any replacement-selection within one block. In term of replacement-selection within the entire tournament tree, there are only a few of cache miss when it switches between blocks.

3.9.4 Implementation

We randomly generate 1 million records of the entire size of 54Mb within one file. It needs about 4.5 seconds to get it sorted. And if we use multi-thread, it needs 6.5 seconds. It says that the switches between threads needs time and cause more cache miss. In term of one thread, it needs 0.5s to input, 1.0s to sort and almost 3.0s to output the results. Because, we do not load records' content in memory. We need to read the records' content in the input file into memory then write it into output file. The file seeking happens a lot when output. We try to use a file cache to improve it. The cache is used to pre-fetch the records from file while reading some records. It gets a good result while inputting, but costs more time while reading from input for output. Because, while inputting, records are read sequentially. But when output, records are read only once, and records location can't be predicted. So, the best idea for non-parallel disk is to use small and multiple files as input to save the file seeking time.

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 at the beginning of your main CPP file 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 3: 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	2009-02-31	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
				Kevin	File version of Trie, B-Trie for std::string and sfilib::UString	Done
				Kevin	Cache-conscious collision resolution string hash table, faster than linear hash table	Done
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