iZENELib RoadMap 2010

Yingfeng Zhang

January 6, 2010

Contents

1 Overview

This report presents the major directions that *izenelib* is going to make advances in year 2010, but not limited to year 2010. What we are going to do is to reach those aims continuously. Besides, during the maintainence of *izenelib*, one of the most important principle is:

Whenever we have some components to be shared, we should refactor it to a common library to be reused.

So, this library should be attentioned and contributed by all team members.

2 AM

2.1 External Memory AM

2.1.1 Enhancements

We have accumulated several external memory data structures. The major problems come from four aspects:

- Random writing performance can not be guaranteed, especially when the data files become larger. Although some merge-based *AM* components have been provided, they have a coarse granularity and are easily blocked during merging.
- Data files might reach up to a huge size, therefore the IO performance could be slowered with the increasing of data size.
- Memory allocation is not based on allocator, so that it is hard to deal with memory fragmentation problem.
- Many locks are used within sources, the concurrency performance has been sacrifice to satisfy the thread-safe.

Accordingly, the enhancements for these problems are:

- Making research on the solution for improving random writing. A possible candidate is the *cache-oblivious* series accessing methods.
- Introducing compression to accessing methods. This work has something to do with the *Compression* section of this report. However, it might not be possible to provide a general compression framework together with the existing *izenelib::am* components, because search compressed data might require a totally new data structure.
- All memory management of newly added data structures will use *allocator* to manage its memory, so that either our own *MemoryManager* or other allocators having different policies could be easily configured.
- Making research on whether it is possible to provide the components for atomic IO.

2.1.2 New features

These are new features that might be helpful to our projects:

- Accessing methods over *PDM—Parallel Disk Model*. In that case, we could deal with the situation where multiple hard disks exist, it is possible when huge amount of data are going to be stored without concuming too much CPU.

- Exact matching *AM*, such as prefix matching, substring matching, wildcard matching. Besides, we might have special requirements, for example: most of the existing exact matching *AMs* do not provide ability to store key-value pairs, while we do.
- Approximate matching *AM*. Existing *Query-correction SubManager* has already provided such a kind of ability. We need to put it to *izenelib::am* after refactoring, so that only the part of approximate matching is put to library.

2.2 In Memory AM

- Search-relevant
  1. Ordered

     The existing *B-Tree* based search trees can satisfy most of the requirements. However, it has two problems:

     – Memory consumption is larger.

     – Concurrent version is not effective because of the existence of lock.

     Some candidates, such as *SkipList*, might be chosen to implement the *lock-free* container, with a less memory consumption.
  2. Un-ordered

     – Hash—The existing *rde::hash* can satisfy most of the requirements, but it has the same memory consumption problem. We have *CCCHash* as a candidate for this problem, but it can not work under large data size. We are going to fix it this year.

     – Minimum Perfect Hashing Function—General *MPHF* is not that valuable for us, because it can not tell whether a certain key exists or not, and therefore can not serve as the base of some search structure. This has been resolved by some newer *MPHF*, such as *monotone MPHF*. We might also add this utility to *izenelib*.
- Threaded collections building block. Such kinds of collections suit includes the frequently used data containers that are thread safe. We might looking forward to those building blocks based on *lock-free* techniques instead of the *lock-based* ones for higher efficiency.

3 IR

3.1 Index

A light-weight inverted index implementation is expected to be provided, together with the searching utilities. We are going to test the cutting-edge techniques on it, for example, on the indexing side:

- Sorting documents not according to their ids, but *prior scores*, or *impact-order*.
- State-of-the-art posting compression.
- Optimizations for *phrase search*, or *proximity based scoring*.

What's more, an elaborated vocabulary is going to be designed, so that memory consumption, accessing speed, and wildcard query can all be satisfied.

On the query side:

- Several optimizations for query evaluation are going to be provided, different kinds of which cooresponds to the sorting order of documents in postings.
- Integrate existing *RankingManager* to *izenelib::ir*, so that it can be reused by both *IndexManager*,and the lightweight index.

3.2 LA

LAManager is not going to be a stand-alone library, it will be put into *izenelib::ir*.

4 Utilities

4.1 Compression

We need a compression framework so that it can:

- Compress/decompress unicode strings fast and effectively.
- Have order preserving as a policy when compressing/decompressing strings.
- Try to combine together with posting compression in index of *izenelib::ir*. It means compressing/decompressing a series of integers is also a possible candidate interface.

In addition, it's better to integrate such a compression framework into the *serialization* framework, so that we can seamlessly store the compressed data into existing *izenelib::am* components.

4.2 Memory

- Memory pool based *MemoryManager*, we also need to encapsulate it to be compatible the *std::allocator*, so that it can be easily passed into other components as a policy.
- Providing solution for a general *MemoryManager*, we hope to reach these aims:
  1. High performance
  2. Little memory fragmentation

3. Thread-safe or lock-free
4. High scalability

Essencially speaking, they conflict with each other. We have to make a good trade off among them.

4.3 Misc

- Utilities from *wiselib* will be transferred to *izenelib*, such as *UString*, to reduce the maintainence cost.
- Adding small tools for system development, such as *crash reporting system.*

5 Net

izenelib::net is going to provide distributed solutions for all projects. Currently, only *MessageFramework* is included. These components are possibly required in future:

- Distributed framework—It can satisfy data mining tasks over tens of machines, which can satisfy our requirements. All of our distributed algorithms will base on this framework to simplify the development. Two essentials are required for this library:
  - A distributed computing model which is easily used, such as *Map/Reduce.*
  - A configurable file system, which user can choose to store data over distributed file system or locally.
- Multiplexing RPC—*MessageFramework* is so heavy:we must provide a *Controller* to run seperatedly. In many situations, we just want an *end-to-end rpc* mechanism. *Multiplexing RPC* is in fact a lightweight *MessageFrameWork.*