# Technical Report of SDB-v2

Peisheng Wang

March 27, 2009

**Abstract**

SDB(Sequential Database), including ordered SDB and unordered SDB, has become very important infrastructure for our many applications.

To have a better SDB, beside B-tree, we also tried other data structures like btree variants, skiplist and hash in cache conscious/oblivious and self adjusting pespective.

SDB/btree is re-implemented to be more cache conscious. B*-btree is adopted to improve disk space efficiency. At the same time, the overflow mechanism and cache mechanism are also improved. Compared to previous btree, the speed of reading, writing and sequential access are improved by 25%, and deleting are twice faster, its file size is also reduced by about 40% in a given test.

SDB/hash, built on static bucket-chained hash, is very efficient. It outperformed SDB /btree and BerkeleyDB/hash in terms of reading, writing, deleting or sequential access. It is about 3 time faster than SDB/btree and twice faster than BerkeleyDB/hash. It can also compete with the hash of tokyo cabinet(TC/hash). Without using mmap, SDB/hash has worse performance than TC/hash in writing, but it didn't perform worse and somtimes can even be better in reading when well tuned. Moreover, it is more cache conscious. It showed better performance than TC/hash in both reading and writing for highly skewed sequence test. It can be also easily adapted to self adjusting data structrue by recording the visted frequency of the items in chained bucket.

Unfortunately, skip list was less efficient than B-tree when reading/writing, due to its less reference locality, However, it doesn't perform worse in deletion and sequential access.

SDB-v2 is also made policy based, SDB/btree, SDB/hash, SDB/skiplist and can be used in the same way through template template parameter. Tokyo Cabinet is also wrapped into SDB-v2 so that any type of keyType and ValueType are avaible for it.

IndexSDB(built on ordered SDB), as a vital component for IndexManager, was also improved. It is much faster and use less disk space.

At the appendix, a guideline of how to use SDB and tuning points are given.

## Contents

## 1 Document History

| Date | Author | Description |
|---|---|---|
| 2009-03-18 | Peisheng Wang | Add SDB/hash |
| 2009-03-20 | Peisheng Wang | Add new data structrue cc-b*-btree and wrap Tokyo cabinet |
| 2009-03-25 | Peisheng Wang | first full version |

## 2 Project goal

Our goal of this project is to buid a better SDB, named as SDB-v2.

SDB-v1, based on btree, keeps data in order, but the ordered of data are not needed. For example, when large amount of images from web are collected for fast retrieval, ordered sequential access is unnecessary. For convenience, we name them ordered SDB and unordered SDB respectively.

Cache Conscious/Oblivious and Self Adjusting property have become more and more important for designing advanced high efficient data structure. we'd like to make SDB-v2 Cache Conscious/Oblivous or self adjusting.

Table 1: storage hierarchies

| To Where | Cycles |
|---|---|
| Register | $\leq 1$ |
| L1 | about 3 |
| L2 | about 14 |
| memory | about 240 |
| disk | >>memory |
| network | >>disk |

## 3 Introduction

SDB-v1, based on B-btree, is a very efficient ordered SDB[5], which is more efficient and easy to be used than BekerleyDB in some applications. Unlike usual DBs, SDB-v1 doesn't store data(key/value pair) in memory in form of DBT(DbObj, binary form of key/value), . Instead, it only transforms data into or from DBT when writing to or reading from disk, thus it reduce comparing cost for user-defined key compared functions.

Beside B-tree, Skip list[7] is another canadiate for constructing ordered SDB. As [7] said that, it is simpler and uses less memory space. It can be also used in k-dimenesion range search, see [3] . But when processing large data, evidence [1] demonstrates that it has worse real-world performance and require more space than B-tree, for its memory locality and other issues. For extern memory access, [8] show that self adjusting skip list(SASL), as self adjusting data structure that keeps the most frequent items on top level, can reach static optimicalty in theory and can be better than btree if the sequence of queries is highly skewed and changes over times. But it is just theoretical result with constrained assumption that the self adjusting didn't affect I/Os.

For unordered SDB, hash would be a better canadiate than btree and skip list, since it doesn't requre the order.

The next we will first introduce Cache Conscious/Oblivious and Self Adjusting propeties of advanced data structre.

### 3.1 Cache Conscious/Oblivious Data structure

The storage hierarchies of modern computers are becomming increasingly "steep", ranging from faster registers and on-chip cache down to relatively slow disks and networks. As the speed of processors is increasing more quickly than the speed of memory and disk, the disparity between the various levels of memory hierarchy(including disk or networks) is growing, see table 1.

The distinguishing feature of multilevel storage hierarchies is that data transfers are done in blocks in order to amortize the cost of transfer. The amortization only works when each transfer contains many pieces of data to be used.

Databases development has brought many mature cache solutions to bridge the gap between memory and disk. Cache Conscious/Oblivious properties, to narrow the gap between CPU and memory, are becoming more and more important.

[1]http://resnet.uoregon.edu/~gurney_j/jmpc/skiplist.html

Cache Conscious/Oblivious algorithms have been advanced as a way of circumventing some of the difficulties of optimizing applica- tions to take advantage of the memory hierarchy of mod- ern microprocessors.

For trying to make use of CPU memory, Cache Oblivious is passive while cache conscious is postitve. Cache Conscious need to be well tuned according to cache parametes. Cache Oblivous is independent of parameters of meory hierarchy. Cahce Conscious ofter use clustering, spliting,reordering techique to reorganize the data structure.

Cache Oblivious are based on the divide-and-conquer paradigm each division step creates sub-problems of smaller size, and when the working set of a sub-problem fits in some level of the memory hierarchy, the computations in that sub-problem can be executed without suffering capacity misses at that level.

Cache Conscious/Oblivious data structure can also be used to enhance Database design, and Cache Oblivious btree has proved to be outperform traditional btree for DB.

The next we will introduce Cache mechanism briefly.

#### 3.1.1 Cache

Reference locality is the reason why Cache works [4]. The basic principle for Cache is to reduce cache misses as much as possible by exploiting best prefetching and replacement policies.Thus, the objective of design Cache Oblvious data structure is to mantain locality of reference, which means that memory accessed are clustered in times and space. Cache Conscious/Oblivious data structure is to boost enhancement of CPU cache L1 and L2. Mostly we don't directly control L1 and L2, instead we try to make use of it by make data compact in memory by trying to use array instead of pointer.

### 3.2 Self Adjusting Data Structure

Self-adjusting data structures will change even as datas are accessed. For self adjusting single list, every item visted will be moved to the front. Ideally, the second access takes O(1) time, even if it takes longer to access the data for the first time around. Self adjusting is to enchance reference locality for sequential access. But sometimes adjusting process itself often takes a lot of time, and for this case, we'd better conduct self adjusting periodically or when idle.

## 4 Previous Work

There have been many existing data structures can be used for SDB.

### 4.1 Tree

SDB-v1 is based on B-tree. B-tree is balanced tree that most commonly used by database systems for implementing index structures. B-trees are optimized for using the minimum number of disk operations for large data structures. It's a tree data structure that keeps sorted data and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and file systems, like BerkeleyDB,QDBM,Tokyo Cabinet and others DBs. Btree aslo has many variants, like $B+$ tree, $B^*-$ btree, $B^{\#}$ tree, and others.

[2] T-tree is a type of binary tree data structure that is used by main-memory databases, such as DataBlitz, eXtremeDB, MySQL Cluster, Oracle TimesTen and KairosMobileLite and by some operating system kernels such as, for example, Ustring - a kernel of Jari operating system(actually it uses a T*-tree - one of variations of original T-tree semantically very similar to B*-tree).A T-tree is a balanced index tree data structure optimized for cases where both the index and the actual data are fully kept in memory, just as a B-tree is an index structure optimized for storage on block oriented external storage devices like hard disks. T-trees seek to gain the performance benefits of in-memory tree tructures such as AVL trees while avoiding the large storage space overhead which is common to them.

[3] There is also dancing tree, which is a tree data structure invented by Hans Reiser. It is used by the Reiser4 file system. As opposed to self-balancing binary search trees that attempt to keep their nodes balanced at all times, dancing trees only balance their nodes when flushing data to a disk (either because of memory constraints or because a transaction has completed). We can conduct self adjuting strategy as dancing tree.

[4] R-trees are tree data structures that are similar to B-trees, but are used for spatial access methods i.e., for indexing multi-dimensional information; for example, the (X, Y) coordinates of geographical data. A common real-world usage for an R-tree might be: "Find all museums within 2 miles (3.2 km) of my current location". The data structure splits space with hierarchically nested, and possibly overlapping, minimum bounding rectangles (MBRs, otherwise known as bounding boxes, i.e. "rectangle", what the "R" in R-tree stands for).Each node of an R-tree has a variable number of entries (up to some pre-defined maximum). Each entry within a non-leaf node stores two pieces of data: a way of identifying a child node, and the bounding box of all entries within this child node.

[5] Kd-tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. kd-trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). kd-trees are a special case of BSP trees.

## 4.2 Cache Oblivious B-tree

Cache Oblivious B-tree, is the data structures that are independent of the parameters of the memory hierarchy, e.g., the number of memory levels, the block-transfer size at each level, and the relative speeds of memory levels. The performance is analyzed in terms of the number of memory transfers between two memory levels with an arbitrary block-transfer size of B; this analysis can then be applied to every adjacent pair of levels in a multilevel memory hierarchy. Both search trees match the optimal search bound of $\Theta(1 + \log_{B+1} N)$ memory transfers. This bound is also achieved by the classic B-tree data structure on a two-level memory hierarchy with a known block-transfer size B. The first search tree supports insertions and deletions in $\Theta(1 + \log_{B+1} N)$ amortized memory

[2]http://en.wikipedia.org/wiki/T_tree
[3]http://en.wikipedia.org/wiki/Dancing_tree
[4]http://en.wikipedia.org/wiki/R_tree
[5]http://en.wikipedia.org/wiki/Kd_tree

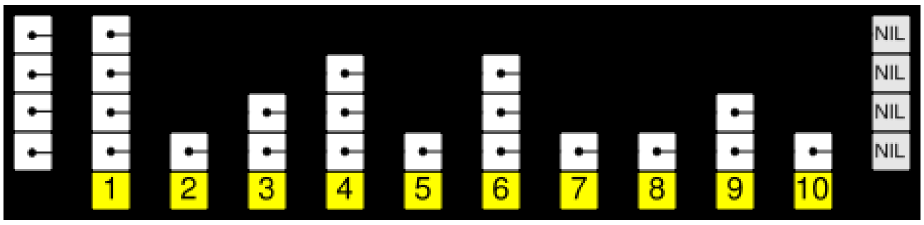

Figure 1: self adjusting Skip list

transfers, which matches the B-tree's worst-case bounds. The second search tree supports scanning S consecutive elements optimally in $\Theta(1 + S/B)$ memory transfers and supports insertions and deletions in $\Theta(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ amortized memory transfers, matching the performance of the B-tree for $B = \Omega(\log N \log\log N)$.

## 4.3 Skip List

Skip list [7] was invented by William Pugh. It is a probabilistic data structure, based on multiple parallel, sorted linked lists, with efficiency comparable to a binary search tree ($O(logn)$ average time for most operations).

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer $i$ appears in layer $i + 1$ with some fixed probability $p$ (two commonly-used values for $p$ are $1/2$ or $1/4$). On average, each element appears in $1/(1 - p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in $\log_{1/p} n$ lists. A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is $1/p$, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list. Therefore, the total expected cost of a search is $log_{1/p}n/p$, which is $\mathcal{O}(\log n)$, when $p$ is a constant. By choosing different values of $p$, it is possible to trade search costs against storage costs.

And it has also many variants, including Deterministic Skip List and Self Adjusting Skip List.

### 4.3.1 Deterministic Skip List

Determinated skip list can keep SEARCH, INSERT, and DELETE in logarithmic time in the worst case.

The basic idea of determinated skip list is to insist that between any pair of elements above a given height are a small number of elements of precisely that height. The desired behaviour can be achieved by either using some extra space for pointers, or by adding the constraint that the physical sizes of the nodes be exponentially increasing. The first approach leads to simpler code,

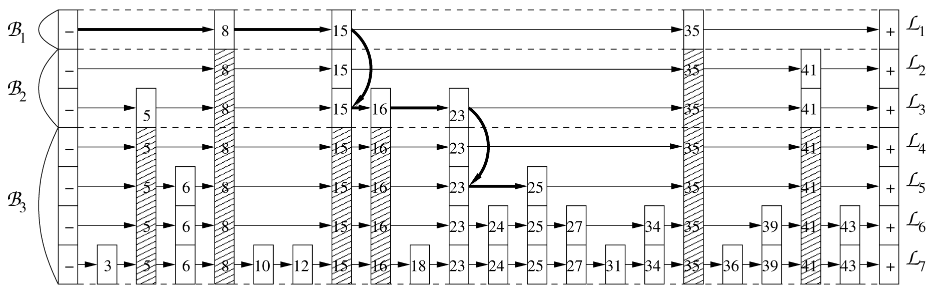

Figure 2: Skip list

whereas the second is ideally suited to a buddy system of memory allocation.It is competitive in terms of time and space with balanced tree schemes, it is also inherently simpler. A vey implementation with sentinel of deterministic skip list is also given in [1]

### 4.3.2 Self Adjusting Skip List

Like spray tree, self-adjusting skip list(SASL)[8] is self adjusting data structure, see figure 2. It will contain random subsets of items in successive levels for balance, and will force frequently accessed items to reside in higher levles to facilitate their future retrieval.

## 4.4 Hash

[6] A hash table, or a hash map, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g., a person's name), find the corresponding value (e.g., that person's telephone number). It works by transforming the key using a hash function into a hash value, a number that is used as an index in an array to locate the desired location ("bucket") where the values should be.

Hash tables support the efficient lookup, insertion and deletion of elements in constant time on average (O(1)) in the number of accessed memory cells); this bound is independent of the number of elements stored, but assumes that a maximal load factor is not exceeded.

Since the intent of hashing is to create unique hash value (or location) for each key, when the hash function produces the same hash value for multiple keys, an alternate location must be determined. The process of finding an alternate location is called collision resolution. A collision resolution strategy ensures future key lookup operations return the correct, respective records. There are a number of collision resolution techniques like open addressing, chaining, but the most popular are open addressing and chaining. The next we will introduce Cache Conscious Collision Resolution

### 4.4.1 Cache Conscious Collision Resolution

A chained hash table is a simple and flexible data structure but it does not make good use of CPU cache. This is especially the case when variable-length strings

[6] http://en.wikipedia.org/wiki/Hash_table

Table 2: file header

| |
|---|
| magic |
| directorySize |
| bucketSize |
| cacheSize |
| height |
| numItem |
| numNodes |
| long *bucketAddress |

are used as keys. When a collision occurs, a linked list is traversed which can attract a surplus of cache-misses. In addition, linked lists will waste an excessive amount of space due to pointers and memory allocation overheads. We can address these issues by replacing linked lists with dynamic arrays, forming a cache Conscious hash table known as an array hash.[2]

When a string is hashed to a slot, it is appended to the end of the dynamic array that is assigned to that slot. In this manner, nodes and pointers are eliminated and strings are accessed in a sequential manner, promoting good use of cache. The array hash was also shown to be substantially faster and more compact than a chained hash table.

## 4.5 Sequence Accesses

Strings algorithm and data structures generally support operations individually and are designed for attaining high efficiency in worst case. However,there is also need for dictionary solutions that are efficient on an entire sequence of strings transactions and possibly adapt themselves to a time-varying distribution of operation. And amortized analysis are used to seek solutions for those problems. Usually worst-case cost over a sequence of operations may be far less than sum of of the wors-costs of single operations in the sequence.

# 5 SDB Based on Skiplist

From [6], Skiplist is equivalent to btree. There are many things in common between btree and skiplist when designing SDB. We will use the same read/write policy, thread-safe policy.

## 5.1 Design and implementation

### 5.1.1 Page Format

The file header page contains the information of SDB. It is stored at the end of file. It will be loaded firstly when opening an existing DB data file.

- Magic: It is set to 0x061561 and used to determinate if the data file has been corrupted. If a file already exists, it will read magic from the file head and verify if it is equal to 0x061561. If not, error returns.

Table 3: Page Format

| |
|---|
| height |
| rightNode address |
| ... |
| rightNode address |
| key/value size |
| key/value |
| overflow address |

- directorySize: When data scale is small, too large directorySize will waste disk space. When data scale is large, too small directorySize will lead to low efficiency.
- bucketSize: The bucketSize is better set to multiples of 8192.
- numItems: The number of items in the hash.
- bucketAddress: The offset of the header node in the file.

Normally, we will treat a node in skiplist as a page. See figure 1, each note contains member **height**, and number **height** pointers to the next nodes. Each skip node was represented as a page in disk, see table 2. We set a **maxDataSize** for these nodes. When it exceeds **maxDataSize**, overflowing occurs,one or more overflowing pages with the same size as skipnode page will be allocated at the end of the file to hold the extra data. The content in the overflow page is just binary, which is the same as btree and BekerleyDB.

```
class SkipNode{
        DataType element;//DataType contains key-value
            pair.
        SkipNode* right[MAX_LEVEL];
        long fpos; //its offset in disk
        long rfpos[MAX_LEVEL];//the offsets of the right
            nodes.
        int height; //its height, by coin flipping if
            normal skip list.

        bool isDirty;
}
```

Since there are too many pointer, it will reduce local reference and lead to low efficiency. so a better design should take cache conscious into consideration. Moreover each page only contains one node and small page would lead to more **fseek** in file, we introduce **Memory Cluster**.

### 5.1.2 Memory Cluester

To reduces I/Os overhead, we map serveral sequential nodes into one memory block. We only read or write the node from or to the memory block. And we read or write one memory block from or to the disk at a time.

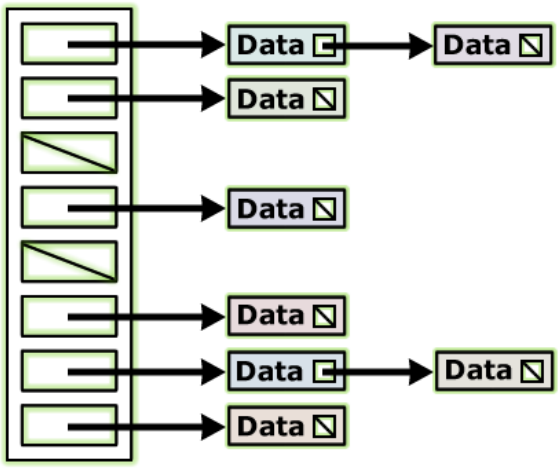

Figure 3: hash-table,**data** is a buket chained

### 5.1.3 Read/Write Policy

We use the same read and write policy as sdb/btree. Everytime we will read or write a page through its **file position**. We only read a page when we acturaly need it, and only write back a page when it is dirty.

### 5.1.4 Cache Mechanism

Cache size means the active pages in memory. When the active nodes number exceeds cache size, it writes back the dirty pages to disk and unload the nodes. Unloading process stops until the active nodes number is within given boudary. Since all nodes are linked in a list, we will use LRU for cache replacement policy.

## 6 SDB Based on Cache Conscious Hash

The hash we use is array hash with cache-conscious collision resolution, see figure 3. When a item(key/value pair) is hashed to a slot, it is appended to the end of the dynamic array that is assigned to that slot. In this manner, nodes and pointers are eliminated and items are accessed in a sequential manner, which promoting good use of cache.

But for SDB, the dynamic array binding to a slot can't be too large. When the directory size is fixed and data scale is very large, the array will grow to very large. Therefore we still introduce the concept of page with fixed size and a dynamic array can be represented with several chained pages.

Although the items of collision are stored sequentially in chained buckets, it will be cache conscious when the buckets are set large enough. Amazingly, it is much faster than linear hash, and btree.

Table 4: key/value pair form in page

| ksize | vsize | key(char*) | value(char*) |
|---|---|---|---|

## 6.1 Design

### 6.1.1 Page

A page is used to store the binary forms items of the collision sequentially. When a page is full, a new page will be allocated for new items of collision. So for each page, there are also two variables to indicate the **num** of collison items within it and the **file position** of nex page.

### 6.1.2 Key/Value Pair Presentation

SDB hash is to store any types of key/value pair. We will transform both key and value into DBT(binary form of key/value pair, with **size** and **char***), and any items are presented as table 4.

### 6.1.3 Read and Write Policy

We use the same read and write policy as sdb/btree. Everytime we will read or write a page through its **file position**. We only read a page when we acturaly need it, and only write a page that is dirty and when **flush** method is called.

### 6.1.4 Cache Mechanism

As is the same as SDB/btree and SDB/skiplist, cache size means active page number in memory. When the active nodes number exceeds cache size, it will writes back the dirty pages to disk and unload the nodes. Unloading stops until the active nodes number is within given boudary. Note that, for efficiency, we unload the tailing bucket in the bucket chain first. Note that the items in the buck chain can only be accessed sequentially. So the front bucket will be access more frequently and viewed more important than the bucket after it.

### 6.1.5 Overflow Mechanism

There is no overflowing for SDB/hash. When a page is not a enough for the new item, A new page will be allocated. Of course the precondition is that the size of inserting items would not exceed the bucket size. And when we want to delete an item, we only set a flag that it is deleted. When we udpdate an item, if the new value size is equal to old one, it will be updated where it is, otherwise we will delete the old items and insert the new one at the end of the bucket chain. Too many updates will cause the disk space waste and low efficiency. For this case, padding may be better solution.

### 6.1.6 Concurrency Mechanism

Read/write lock will be used the same as SDB-v1. And everytime, we only need to lock a bucket-chain, the infected bucket maybe need write lock when there is modification, and the buckets in front of it only need read lock.

Table 5: page format

| |
|---|
| num |
| long nextfpos |
| buffer for key/vaue pairs |

Table 6: file header

| |
|---|
| magic |
| directorySize |
| bucketSize |
| cacheSize |
| height |
| numItem |
| numNodes |
| long bucketAddress[directorySize] |

#### 6.1.7 Self Adjusting Mechanism

For it is chained bucket, we can implement self adjusting property by moving bucket that most frequent visted to the front. The items sequence in a bucket can also be reorganized accordint to their visited frequency. The self adjusting policy should be implemented when idle or as dancing tree does.

## 6.2 Implementaion

### 6.2.1 Page Format

A page format is presented as table 5.

### 6.2.2 File Header

The file header contains the information of SDB. It is stored at the end of file. It will be loaded firstly when opening an existing DB data file.

- Magic: It is set to 0x061561 and used to determinate if the data file has been corrupted. If a file already exists, it will read magic from the file head and verify if it is equal to 0x061561. If not, error returns.
- directorySize: When data scale is small, too large directorySize will waste disk space. When data scale is large, too small directorySize will lead to low efficiency.
- bucketSize: The bucketSize is better set to multiples of 1024.
- numItems: The number of items in the hash.
- bucketAddress: The offset of the header node in the file.

### 6.2.3 Cache Mechanism

An actvie page was represented as an bucket_chain object, and it has attribute **level**, and the **level** of its next bucket_chain has increament by 1. When active pages in memory exceed cache size, we call the following route:

```
//flush Cache will be called in search routing.
void flushCache()
{
  if(active page num >cache size){
    foreach page in memory{
      cacheMap_.insert(level, page);//cacheMap_ is a
          STL map structure.
      foreach page staring from the front in cachMap_
      {
        unload(it);
        if page is dirty
          write back the page;
         if( active number lower than given bound)
          stop unload
      }
 }
```

The cache mechanism of B-tree is almost the same. But it call **commit()** to write dirty pages to disk. To reduce IO overheading, we can just write back the dirty nodes which are to be unloaded within **flushCache_()**.

### 6.2.4 SDBCursor

It also introduces the concept of SDBCursor for range search and query.

```
//the second member is the position in bucket_chain
    buffer for a key/value pair.
typedefpair <bucket_chain*, char*> SDBCursor
```

# 7 SDB based on Cache conscious B-btree

We have re-implement SDB/btree to make it more cache conscious. At the same time, we used b*-btree to improve disk space efficiency and its overflow mechanism, cache mechanism are improved.

## 7.1 b*-btree

A B*-tree is a B-tree in which each node is at least 2/3 full (in-stead of just 1/2 full). B*-tree insertion employs a local redistribution scheme to delay splitting until 2 sibling nodes are full. Then the 2 nodes are divided into 3, each 2/3 full. This scheme guarantees that storage utilization is at least 66%, while requiring ing only moderate adjustment of the main-tenance algorithms. It should be pointed out that increasing storage utilization has the side effect of speeding up the search since the height of the resulting tree is smaller.

For implementation convience and easy future matainess, we only using the delay splittiing for leaf nodes. For most of the nodes are in the leaves when degree(acturally we use degree at least >16) is not small, this strategy is preferred.

However, for ascending insertion, b*-btree is much slower than b-btre. Compared to b-tree, the implementation of b*-btree only differs in splilting. And we make it policy based in implementation.

```
bool insert(node, key){
 child = node->search(key);
 if (child not leafnode) or (not delaysplit)
     split(); //split a full node to two half-full nodes.
 else
     delaySplit();//if the targeted node is full, then
         insert it into adjacent node,
                    //if the adjacent node is also full,
                        then split the two adjacent
                    //nodes into 3 2/3 full nodes.
}
```

## 7.2 overflow mechanism

The overflow mechanism is also improved. Now all items in a btree node share the node page and overflow page.

- It can support any size of key/value pair, the only constraint is that its size cant be too large for a overflow page to hold it.
- To have less parameters, we set overflowpage size to the page size. When overflow occurs, we would allocate several sequential overflow pages at the end of file to hold it.
- The overflow mechanism is more effective than SDB/btree_v1. Now, we dont have maxDataSize parameter, which is used to determine when overflow occur in SDB/btree_v1 and it would cause more than one time- consuming fseek operations. Instead, for cc-b*-btree, we only need at most one more *fseek* operation when overflow occur and use less disk space.
- For update operation, if the size of on btree node still within already allocated overflow pages, then no extra overflow pages will allocated. otherwise, new sequential overflow pages will be allocated and the old overflow pages will be never used again. Of couse too many such update or delete opetation would waste disk space, but we can avoid it by reorganizing the data file when we monitor disk space usage is low efficiency.

## 7.3 Cache mechanism

Cache size means the number of active page in memory. When cache is full, i.e. active B-tree nodes in memory exceeds cache size, we will eliminate some nodes. For any nodes can are reached from root node, we view that top level nodes are more important than low level nodes. We use queue to implement replacement policy.

```
flushCache(){
   queue q ;
   int popNum = 0 ;
   int escapeNum = 0 ; // the top nodes will be escaped
       from unloading.
   q.push(root)
   while (!q.empty()) {
     popnode = q.pop() ;
     if ( popNum > escapeNum)
     popNum>unload() ;
     foreachchild of popNode;
       q.push(child) ;
    }
}
```

## 7.4 SDBCursor

It also introduces the concept of DBCursor for range search and query.

```
   typedefpair <BTreeNode, elementNO> SDBCursor.
```

## 8 SDB based on Tokyo Cabinet

[7] Tokyo Cabinet is a library of routines for managing a database. The database is a simple data file containing records, each recored is a pair of a key and a value. Every key and value is serial bytes with variable length. Both binary data and character string can be used as a key and a value. There is neither concept of data tables nor data types. Records are organized in hash table, B+ tree, or fixed-length array.

However, for our SDB, we can use any types of KeyType and ValueType, and we wrap for it as following: (for DbObjPtr, write_image(), read_image(), see [9]).

```
bool insert(const KeyType& key, const ValueType& value) {
  DbObjPtr ptr, ptr1;
  ptr.reset(new DbObj);
  ptr1.reset(new DbObj);
  write_image(key, ptr);
  write_image(key, ptr1);

  //tokyo cabinet's insertion method
  return tchdbputkeep(hdb_, ptr->getData(), ptr->getSize
      (), ptr1->getData(), ptr1->getSize());
}

ValueType* find(const KeyType & key) {
  DbObjPtr ptr, ptr1;
  ptr.reset(new DbObj);
```

[7]http://tokyocabinet.sourceforge.net/index.html

```
  write_image(key, ptr);

  void* value =NULL;
  int sp;

 //tokyo cabinet's get method
  value = tchdbget(hdb_, (void*)(ptr->getData()), ptr->
      getSize(), &sp);
  if( !value )return NULL;
  else {
     ptr1.reset(new DbObj(value, sp));
     ValueType *val = new ValueType;
     read_image(*val, ptr1);
     return val;
  }
}
```

## 9 SDB-V2

SDB v2.0 is built on izenelib [9], where all the underlying data structure of SDB like B-tree, skip list, and sdb-hash are placed.

### 9.1 Policy Based

SDB was designed to be policy-based, it can use either sdb/btree, sdb/hash, tokyo-cabinet/hash, tokyo-cabinet/btree, skiplist or other data structures as underlying container in the same way. And SequentialDB is declared as following:

```
template<
         typename KeyType,
         typename ValueType=NullType,
         typename LockType =NullLock,
         typename ContainerType=izenelib::am::sdb_btree<
             KeyType, ValueType, LockType>,
         typename Alloc=std::allocator<DataType<KeyType,
             ValueType> >
         >
class SequentialDB {}
```

## 10 IndexSDB

IndexerManager stores $TermID/vector\langle DocID\rangle$ as $Key/Value$ pair. Usually the size of Value can be very small or very large. And it will be of low efficiency and waste a lot of disk space if we use original SDB by setting very large page-size to avoid too much overflowing. Instead, we use several sub keys to represent one key and original value, $vector\langle ElementType\rangle$, are scattered to serveral sequential nodes, see [5].

```
//wrapper for KeyType
template<class KeyType> struct iKeyType {
        KeyType key;
        unsigned int offset;
}

template<
   class KeyType,
   class ElementType,
   class LockType=NullLock
>
class IndexSDB {
   typedef iKeyType<KeyType> myKeyType;
   typedef std::vector<ElementType> myValueType;

   SequentialDB<myKeyType, myValueType, LockType> _sdb;
}
```

IndexSDB also provided all kindes of queries, including range queries, prefix query, and even substring match query.

### 10.1 substring match

For substring match query, we store $suffix\ vector\langle word\rangle$ for every word into IndexSDB, and a keys has several suffices, and a suffices has servral responding words.

## 11 Experimentation and Performance

In order to do comparision testing among SDB/btree, SDB/skiplist, SDB/hash, BDB/btree and BDB/hash, two testings are given, one for unique input(asscending and random), the other for skewed words, with many repeatings.

### 11.1 SDB-v1 vs SDB-v2

#### 11.1.1 Test1

Test1 is to calculate processing time (real time) and file size of database. Writing test is to store 1,000,000 records. Reading test is to fetch all of its records through keys. Deleting test is to delete all of the records through keys. Traversal test is to fetch all of the records from the beginning to the end

Platform: Fedora 8, EXT2 file system, Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz , 4096MB RAM. Compilation: gcc 4.1.2 (using -O3), glibc 2.7

Keys: strings in range of ‘00000000’, ‘00000001’, ‘00000002’... ’00999999’

Values: NullType.

Time: Unit of time is seconds.

We insert the items both in ascending and random order

Table 7: DB comparision1

| DB | write | read | traversal | del |
|---|---|---|---|---|
| SDB/btree_v1/rnd | 5.04 | 3.78 | 0.48 | 11.12 |
| SDB/btree_v1/asc | 2.4 | 1.69 | 0.33 | 5.0 |
| SDB/btree_v2/rnd | 3.88 | 2.72 | 0.44 | 4.37 |
| SDB/btree_v2/asc | 1.90 | 1.81 | 0.5 | 7.59 |
| sdb/hash | 1.29 | 0.76 | 0.52 | 1.02 |

Table 8: DB comparision2

| DB | write | read | traversal | del |
|---|---|---|---|---|
| SDB/btree_v1/ | 4.57 | 3.4 | 0.54 | 7.33 |
| SDB/btree_v2 | 3.59 | 2.60 | 0.39 | 3.25 |
| sdb/skiplist | 8.5 | 9.1 | 0.24 | 5.43 |
| sdb/dskiplist | 7.34 | 8.88 | 0.24 | 5.3 |
| sdb/hash | 1.2 | 1.01 | 0.29 | 1.95 |

### 11.1.2 Test2

We read words sequentially from a file, which consists of 1462877 words and the number of unique words are 363529. We take the words as the input key,and value is NullType object. After inserting all of the datas into the SDB, and we call **f**ind(key) with inputing words as key,to test search, then we also we call

```
getValueForward  (count , result , key ) ;
```

to read the records sequentially to do range testing .

The next we call,

```
bool del(key)
```

to do deletion testing.

We did the testing serveral times and get the average time cost for all operations. **dc** and **ic** mean default cache size and infinite cache size respectively.

For self adjusting skip list, if we do promotion and demotion after each search, it will be much slower, for promotion and demotion takes more time than search itself. Maybe promotion and demotion should be executed later.

### 11.1.3 Result Analysis

From table 7 and table 8 we can have those conclusion as following: (Note that, sdb/btree, sdb/skiplist, sdb/hash all have infinite cache size.

- SDB/hash outperform SDB/btree SDB/hash perform best. It is about 4 times faster than btree.
- SDB/btree_v2 outperform SDB/btree_v1. And when we refer SDB/btree, we means SDB/btree_v2.
- Skiplist is slower than btree in reading and writing, while for deleting it is faster than btree. Deterministic skip list is faster than normal skip list in reading and writing.

Table 9: SDB vs TokyoCabinet vs BekeyleyDB: Test1

| DB | write | read | fileSize(megabytes) |
|---|---|---|---|
| SDB/hash | 1.25 | 0.76 | 33.62 |
| TC/hash | 0.83 | 0.67 | 32.52 |
| BDB/hash | 2.68 | 1.73 | 20.93 |
| SDB/btree/asc | 2.01 | 1.82 | 32.48 |
| SDB/btree/rnd | 3.88 | 2.68 | 25.15 |
| TC/btree/asc | 1.03 | 0.93 | 12.32 |
| TC/btree/rnd | 3.67 | 3.61 | 12.78 |
| BDB/btree/asc | 1.02 | 1.81 | 20.39 |
| BDB/btree/rnd | 2.76 | 2.45 | 18.17 |

- For skewed sequential access, SDB/hash, SDB/btree are more cache conscous and more faster.

## 11.2 SDB vs TokyoCabinet vs BekeyleyDB

To thoroughly compare the performance among our SDB, TokeyCabinet and BekeyleyDB, including hash and btree. We do four kinds of testing. Note that, all our SDB are tuned to the best performance, and TokeyCabinet BekeyleyDB all use the infinity cache.

The performance of b-tree relates a lot to the distribution and sequence of inserting keys, we also do ascending testing and random testing.

However, for the keys stored in hashes are unordered, we don't need to test the case that keys are inserted into at random. Test1 and Test2 has the same setup as section 11.1

Note that, **BDB/btree/asc** means BerkeleyDB using btree and inserting kesy are ascending from '00000000', '00000001', '00000002' to '00999999'. **TC/hash/rnd** means Tokyo Cabinet using btree inserting keys are random in range of from '00000000' to '00999999'

### 11.2.1 test1

- Keys: strings in range of '00000000', '00000001', '00000002'... '00999999'
- Value: NullType.
- It includes ascending testing and random testing for ordered DBs.

Conclusion, see table 9.

- TC/hash >SDB/hash >BDB/hash

  For writing, TC/hash is about 33% faster than SDB/hash, SDB/hash is about 2 times faster. For reading, TC/hash is only about 10% faster than SDB/hash, SDB/hash is 2 times faster than BDB/hash.

- TC/btree/asc >BDB/btree/asc >SDB/btree/asc

  BDB/btree/rnd >SDB/btree/rnd >TC/btree/rnd

  For ascending writing,TC/btree has the same efficiency as BDB/btree, BDB/btree is about twice faster than SDB/btree. For ascending reading,

Table 10: SDB vs TokyoCabinet vs BekeyleyDB: Test2

| DB | write | read | fileSize(megabytes) |
|---|---|---|---|
| SDB/hash | 1.02 | 0.89 | 12.80 |
| TC/hash | 1.06 | 0.92 | 12.16 |
| BDB/hash | 3.74 | 2.18 | 20.94 |
| SDB/btree | 3.3 | 2.58 | 15.01 |
| TC/btree | 3.52 | 3.56 | 5.3 |
| BDB/btree | 3.70 | 3.11 | 20.89 |

TC/btree is about twice faster than BDB/btree. And BDB/btree is the same as SDB/btree.

For random writing, BDB/btree is about 25% faster than TC/btree, and TC/btree is almost the same as SDB/btree. For random reading, BDB is about 33% faster than TC/btree, and 5% faster than SDB/btree.

#### 11.2.2 test2

- Keys: It reads words sequentially from a file, which consists of 1462877 words and the number of unique words are 363529. This testing sequence are highly skewed test.
- Values: NullType object.

Conclusion, see table 10.

- SDB/hash >TC/hash >BDB/hash

  For highly skewed sequence, SDB/btree perform a little faster than TC/hash both in reading and writing. TC/hash is about 3 times faster than BD-B/hash
- SDB/btree >BDB/btree >TC/btree

  For writing, SDB/btree is a little faster than TC/btree, and TC/btree is a little faster than BDB/btree. But for reading, SDB/btree is about 20% faster than BDB/btree, and 30% faster than TC/btree.
- TC/hash and TC/btree use least disk space.

#### 11.2.3 test3

- Keys:in range of ‘00000000’,0 , ‘00000001’,1, ‘00000002’,2... ’00999999’,999999 and value is NullType.
- Values: NullType object.

```
struct MyKeyType {
  String key;
  int data;

  //it uses default boost:serialization method to
      implement
```

Table 11: SDB vs TokyoCabinet vs BekeyleyDB: Test3

| DB | write | read |
|---|---|---|
| SDB/hash | 16.36 | 8.82 |
| TC/hash | 9.19 | 9.3 |
| BDB/hash | 17.48 | 13.96 |
| SDB/btree/asc | 10.95 | 9.86 |
| SDB/btree/rnd | 9.19 | 10.26 |
| TC/btree/asc | 135.3 | 8.09 |
| TC/btree/rnd | 415.16 | 8.52 |
| BDB/btree/asc | 28.18 | 28.23 |
| BDB/btree/rnd | 29.27 | 30.2 |

```
    //read_image() and write_image(), and it is slow.

    template<class Archive> void serialize(Archive & ar,
                    const unsigned int version) {
        ar & key;
        ar & data;
    }
    inline int compare(const MyKeyType& other) const {
        return key.compare(other.key);
    }
}
```

Conclusion,see table 11.

- TC/hash >SDB/hash >BDB/hash

  But for reading, TC/hash is only a little faster.

- SDB/btree >BDB/btree >TC/btree

  SDB/btree is about 3 times faster than BDB/btree, and 40 times faster than TC/btree.

### 11.2.4 test4

- Keys: in range of 0,0 , 1,1, 2,2... 999999,999999
- Values: NullType object.

```
struct MyKeyType {
    int key;
    int data;

    inline int compare(const MyKeyType& other) const {
        return key- other.key;
    }
}
```

Table 12: SDB vs TokyoCabinet vs BekeyleyDB: Test4

| DB | write | read |
|---|---|---|
| SDB/hash | 1.02 | 0.56 |
| TC/hash | 0.68 | 0.61 |
| BDB/hash | 7.31 | 3.63 |
| SDB/btree/asc | 0.63 | 0.4 |
| SDB/btree/rnd | 0.82 | 0.69 |
| TC/btree/asc | 4.09 | 0.34 |
| TC/btree/rnd | 46.84 | 0.36 |
| BDB/btree/asc | 1.06 | 1.05 |
| BDB/btree/rnd | 1.09 | 1.06 |

```
//Its serialzation is much faster than boost::
    serialization.
NS_IZENELIB_AM_BEGIN
namespace util {
  template<> inline void read_image<MyKeyType>(MyKeyType&
        dat, const DbObjPtr& ptr) {
      memcpy(&dat, ptr->getData(), sizeof(MyKeyType));
  }
  template<> inline void write_image<MyKeyType >(const
      MyKeyType& dat, DbObjPtr& ptr) {
      ptr->setData(&dat, sizeof(MyKeyType));
  }
}
NS_IZENELIB_AM_END
```

Conclusion,see table 12:

- TC/hash >SDB/hash >BDB/hash

  For writing, TC/hash is the most fastest. For reading, SDB/hash can be faster than TC/hash.

- SDB/btree >BDB/btree >TC/btree

  SDB/btree performed others DBs.

### 11.2.5 conclusion

- For writing, TC/hash >SDB/hash >BDB/hash. But for reading, SDB/hash didn't perform worse. And for high skewed sequence, SDB/hash perform better than TC/hash in both reading and writing.
- When keys are strings,
  - TC/btree >DBD/btree >SDB/btree, when keys are ascending and unique, and TC/btree also use least disk space.
  - BDB/btree >SDB/btree >TC/btree, when keys are random.
  - SDB/btree >BDB/btree >TC/btree, especially for reading, when keys are high skewed sequence.

Table 13: example: BDB vs SDB

| DB | operation1 | operation2 |
|---|---|---|
| times | 1000000 | 3000 |
| BDB/btree | 12.35 | 0.75 |
| SDB/btree_v1 | 7.8 | 0.37 |

- SDB/btree outperform the others when using user defined key-comparision function.
- SDB/btree and hash are more cache conscious, it performs better when keys are skewed sequence.

### 11.3 example

We have tested Sequential DB and Berkeley DB to see if SDB performs better in actural project in izenesoft. There're two operation we need:

- Operation1: Insert a record into the database, the record is in the format of <termID1: unsigned int, termID2: unsigned int, sim: double >
- Operation2: Given a term $ID_1$, get the related terms $ID_2s$ with the top $N$ sim values.

For this example, **KeyType** are as follows, and **DataType** only contains **KeyType**, **ValueType** is NullType.

```
struct myKey{
        int temID1;
        int termId2;
        double simScore;
};
```

After the optimization, the Sequential DB runs faster.

## 12 CacheDB vs SDB

SDB also uses cache to achieve high efficiency, but their cache mechanism is different, CacheDB is outer, while SDB is embedded. For SDB/btree or SD-B/hash, we only reserve high level nodes or bucket in memory for fast access when data scale is very large. And for CacheDB, we reserve most recent or frequent items for repeating visiting. So CacheDB and SDB can complement with each other, CacheDB can be used to cache the recent or frequent items in SDB's low level nodes with large data scale. And when data set is not so large, SDB wth enough cache is enough, no cacheDB needed.

The rule is that, to avoid duplicate items in memory.

## 13 Future works

- The disk space efficiency of SDB/hash and SDB/btree can be improved by take advantage of Tokyo Cabinet.
- SDB/hash can be improved by using mmap.
- Concurrency of SDB can be also improved.

## 14 Conclusion

To have a better SDB, beside B-tree, we also tried other data structures like btree variants, skiplist and hash in cache conscious/oblivious and self adjusting pespective.

SDB/btree is re-implemented to be more cache conscious. B*-btree is adopted to improve disk space efficiency. At the same time, the overflow mechanism and cache mechanism are also improved. Compared to previous btree, the speed of reading, writing and sequential access are improved by 25%, and deleting are twice faster, its file size is also reduced by about 40% in a given test.

SDB/hash, built on static bucket-chained hash, is very efficient. It outperformed SDB /btree and BerkeleyDB/hash in terms of reading, writing, deleting or sequential access. It is about 3 time faster than SDB/btree and BerkeleyDB/hash. It can also compete with the hash of tokyo cabinet(TC/hash). Without using mmap, SDB/hash has worse performance than TC/hash in writing, but it didn't perform worse and somtimes can even better in reading when well tuned. Moreover, it is more cache conscious. It showed better performance than TC/hash in both reading and writing in highly skewed sequence test. It can be also easily adapted to self adjusting data structrue by recording the visted frequency of the items in chained bucket.

Unfortunately, skip list was less efficient than B-tree when reading/writing, due to its less reference locality, However, it doesn't performed worse in deletion and sequential access.

SDB-v2 is also made policy based, SDB/btree, SDB/hash, SDB/skiplist and can be used in the same way through template template parameter. Tokyo Cabinet is also wrapped into SDB-v2 so that any types of keyType and ValueType are avaible.

## References

[1] R. S. J.Ian Munro, Thomas Papadakis. Deterministic skip lists.

[2] K. A. R. Jun Rao. Cache conscious indexing for decision-support in main memory. 1999.

[3] B. G. N. Michael G. Lamoureux. A deterministic skip list for k-dimensional range search. *Acta Informatica*, 41(4-5), January 2005.

[4] W. Peisheng. Technical report for cache. May 2008.

[5] W. Peisheng. Technical report for sequentialdb. Nov. 2008.

[6] S. L. Prosenjit Bose, Karim Douieb. Dynamic optimality for skip lists and b-trees. 2008.

[7] W. Pugh. Skip lists: A probabilistic alternative to balanced trees.

[8] C. Valentina. A data structure for a sequential of string accesses in external memory. *ACM Transactions on Algorithm*, 3(1), February 2007.

[9] P. W. Yinfeng Zhang, Kevin Hu. Technical report for izenelib. 2009.

## A Schedule

| | Miles-tone | Start | Finish | In Charge | Description | Status |
|---|---|---|---|---|---|---|
| 1 | Preparation | 2008-12-08 | 2008-12-12 | Peisheng Wang | Study Pugh'Paper about skiplist and understand skiplist. Study determined skip list's souce code and study source about skipDB, which is like BerkelyDB but using skiplist not b-tree. Study self-adjusted skip list(SASL) | finished. |
| 2 | Design & archi-tect | 2008-12-15 | 2008-12-19 | Peisheng Wang | Trying to build sdb-v2 based on self adjusting skip list. And since skip list is equivalent to btree, we also determine code part of sdb-v1 that can be reused in sdb-v2. Read policy, write back policy, cache pol-icy, thread safe policy, will be the same as sdb-v1. | finished |
| 3 | Implementation of skip list | 2008-12-21 | 2008-01-02 | Peisheng Wang | Finish the framwork. | finished |
| 4 | Improvment and experimen-tation | 2008-01-04 | 2009-01-23 | Peisheng Wang | Compare it with sdb-v1,but skiplist perform worse in reading and writing. | finshed |
| 5 | Make sdb-v2 policy-based, and update TR. | 2009-02-03 | 2009-02-06 | Peisheng Wang | Now B-tree and skiplist are avaible in sdb-v2, update TR by adding how to use sdb-v2. | finshed |
| 6 | Implementation sdb-v2 based on hash | 2009-02-09 | 2009-02-13 | Peisheng Wang | SDB/hash is built on izenelib. SDB/hash's underlying data struc-ture is static bucket chained hash. SDB-v2 is policed based. | finished |
| 7 | Experimentation of SDB/hash | 2009-02-16 | 2009-02-20 | Peisheng Wang | SDB/hash outper-formed others in all aspects, it's about 4 times faster than SDB/btree and 2 times faster than BDB/hash. | finished |
| 8 | Update TR and improve cache mechanism of btree-v1. | 2009-02-23 | 2009-02-27 | Peisheng Wang | Update TR including how to use SDB-v2 and IndexSDB. Cache mechanism of B-tree is improved for random reading/writing. | finished |
| 9 | Implement Cache Con-scious B-tree and thorough | 2009-03-02 | 2009-03-13 | Peisheng Wang | cc-b*-btree outperform previous b-tree by about 25% improve-ment. | finished |

## B How to use sdb-v2?

### B.1 example

For usage of these codes, please see example in source directory:

- t_sdb.cc, policy-based example, btree,skiplist, sdb/hash, sdb/btree,tc/hash, tc/btree can be choosed as underlying data structure.
- t_overflow_btree.cc, test overflow mechanism for btree
- t_mul_sdb.cc, example for multi-thread testing for sdb.
- t_IndexSDB.cc, example for IndexSDB, and now it is much faster than IndexSDB of sdb-v1.
- t_tc.cc, t_sdb1.cc, t_bdb.cc in source/db-test directory, for DBs testing among SDB, Tokyo Cabinet, and BekeleyDB.

### B.2 Process

Now SequentialDB is built on izenelib and policy based. Izenelib has define DataType inside, and it contains *KeyType* and *ValueType*. And if *ValueType* is not given, it will default to *NullType*, see *izenelib/am/concept*.

1. Define *KeyType* and *ValueType*. SequentialDB can support different KeyType and ValueType.

   KeyType and ValueType, can be *int*, *float*,*string*, or user defined data types.

   For ordered SDB(SDB/btree and SDB/skiplist), *KeyType* must have compare function. For primitive types like *int*, *float*,*string*, embedded compare function for them are provided in **izeneLib**. While for user defined types, it must provide compare methods as following:

   ```
   stuct myKeyType{
      int a;
      double d;
      int compare(const myKeyType&otherkey);

      //if use default boost serialization, the
          following method must bet provided.
      template<class Archive> void serialize(Archive& ar
          ,
                              const unsigned int version) {
        ar & a;
        ar & d;
      }
   }

   typedef string myValueType;
   ```

By the way, for unordered SDB, compare function is not needed.

When reading or writing an item to disk, it must be transformed from or into *DBT(or DbObj)* form by *read_image(), write_image()* methods in *izenelib/am/util/Wrapper.h.*

To support different types of *KeyType* and *ValueType* easily, default *read_image(), write_image()* method using *boost::serialization.* are as followlling:

```
template<class T> inline void read_image(T& dat,
    const DbObjPtr ptr) {
  stringstream istr((char*)ptr->getData());
  {
      boost::archive::text_iarchive ia(istr);
      ia & dat;
  }
}

template<class T> inline
void write_image(const T& dat, DbObjPtr ptr) {
    stringstream ostr;
    {
        boost::archive::text_oarchive oa(ostr);
        oa & dat;
    }
    ptr->setData(ostr.str().c_str(), ostr.str().size()
         );
}
```

To use those default *read_image(), write_image()* methods, for user defined KeyType and ValueType, them must also provide *serialize()* method the same way as using *boost::serialization.*

**However, for efficieny, it is better for client to provide their own serialization method for *KeyType* and *ValueType* , mostly it can be mush fast.** For example,

```
//myKey and myValue are user defined and have fixed
    length.
typedef myKey{
    int id;
    float score;
}
typedef myValue{
    char name[10];
}

typedef myKey Key;
typedef myValue Value;

NS_IZENELIB_AM_BEGIN

namespace util {
```

```
template<> inline void read_image<Key>(Key& key ,
    const DbObjPtr& ptr ) {
      memcpy(&key , ptr->getData ( ) , sizeof(Key ) ) ;
}
template<> inline void write_image<Key>(const Key&
     key , DbObjPtr& ptr ) {
      ptr->setData(&key , sizeof(Key ) ) ;
}

template<> inline void read_image<Value>(Value&
    val , const DbObjPtr& ptr ) {
      memcpy(&val , ptr->getData ( ) , sizeof( Value ) ) ;
}
template<> inline void write_image<Value>(const
    Value& val , DbObjPtr& ptr ) {
      ptr->setData(&val , sizeof( Value ) ) ;
}
}
NS_IZENELIB_AM_END
```

2. Determined thread-safe policy and which data strutrue to be used, among btree, skiplist, hash or future ones.

```
typedef sdb_btree<Key, NullType, NullLock>SBTREE; //cc
    -b-btree or cc-b*-btree
typedef tc_hash<Key, NullType, NullLock>TC_HASH; //
    tokyo cabinet hash
typedef sdb_hash<Key, NullType, NullLock>SHASH; //
    cache conscious of hash of SDB
typedef BTreeFile<Key, Value, NullLock> BTF; //btree-
    v1
typedef SkipListFile<Key, NullType, NullLock> SLF; //
    skip list

typedef SequentialDB<Key, NullType,NullLock,TC_HASH>
    SDB_TCHASH;
typedef SequentialDB<Key, NullType,NullLock, SBTREE>
    SDB_BTREE;
typedef SequentialDB<Key, NullType,NullLock,SHASH>
    SDB_HASH;

typedef SequentialDB<Key, NullType, NullLock, BTF>
    SDB_BTF;
typedef SequentialDB<Key, NullType,ReadWriteLock,SLF>
      SDB_SL;
```

3. Initialization

   Then call constructor,ant set configuration and tuning.

   - SDB/btree(btree_v2)

```
SDB_BTREE sdb(indexFile);
sdb.setDegree(degree);
sdb.setPageSize(pageSize);
sdb.setCacheSize(cacheSize);
sdb.open();//sdb must be opened for use.
```

- tokyo cabinet hash

  ```
  SDB_TCHASH sdb(indexFile);
  //if we want to tune tc_hash for performance
  tc_hash tch = sdb.getContainer();\\get container
  TCHDB* ptch = tch->gethandle;\\get handle
  tchdbtune(pthc, ....);\\tuning

  sdb.open();//sdb must be opened for use.
  ```

- sdb_hash,

  ```
  SDB_HASH sdb(indexFile);
  sdb3.setDirectorySize(4096);
  sdb3.setBucketSize(8192);
  sdb3.setCacheSize(cacheSize);
  sdb3.open();
  ```

- SDB/btree_v1,

  ```
  SDB_BT sdb(indexFile);
  sdb.setDegree(degree);
  sdb.setPageSize(maxDataSize);
  sdb.setCacheSize(cacheSize);
  //sdg.setOverFlowPageSize(overflowsize); //if not
      called, use default value.
  sdb.open();//sdb must be opened for use.
  ```

- skiplist,

  ```
  SDB_SL sdb(indexFile);
  sdb.setPageSize(maxDataSize);
  sdb.setCacheSize(cacheSize);
  //sdg.setOverFlowPageSize(overflowsize); //if not
      called, use default value.
  sdb.open();//sdb must be opened for use.
  ```

4. Set configuration and tuning

   - SDB/btree (btree_v2)
     - set degree, degrees from 8 to 16 are preferred. Then the number of maxkeys are 2*degre.
     - set PageSize, now there is no maxDataSize, the pageSize is the actural size for one btree node. 512, 1024, 2048, 4096, 8192 are recommended for use according to degree and average data size.

$$pageSize \approx 2 * degree * average\ data\ size$$

- set cacheSize The bigger CacheSize, the better. But it depends on how much memory is avaible.

  $$cacheSize \approx avaible\ memory/pageSize$$

  We can also call,to adjust cacheSize at runtime.

- SDB/hash
  - set directorySize and bucketSize Usually, the bigger directorySize, the more efficient, but too big directorySize will lead to disk space waste. For bucketSize, it is better to set it as multiples of 1024.
    A better policy is that, guess the data scale firstly, then determine directorySize and bucketSize.

    $$average\ bucket\ chain\ length = fileSize/(directorySize*bucketSize)$$

    The average bucket chain length is should within 2. So it's better for whole data scale within the range of from 0 to 2**directorySize*$\times$ *bucketSize* and *directorySize* should not too small.
  - set CacheSize The same as btree, when cache is full, we unload the buckets from the tailing. The bigger CacheSize, the better. But it depends on how much memory is avaible. CacheSize means the active node number and it can be adjusted at runtime.

    $$cacheSize \approx avaible\ memory/bucketSize$$

  - set overflow page size. There is no overflow for SDB/hash, the constrain is that maxDataSize can't not exceed bucket size.
- SDB/btree_v1
  - set degree //degree of 16, 32, 64,128,512... are recommended. When items are random, degree 16$\tilde{3}$2 are the best. When items are ascending or descending, big degree are recommended, like 128,256,512,... or even bigger.
  - set pageSize Note that, now *DataType* can be any size. Although small maxDataSize would save disk space, it is better to set appropriate *maxDataSize* so that the most item not exceed *maxDataSize*,for it will lead to efficiency decline.
  - set CacheSize The bigger CacheSize, the better. But it depends on how much memory is avaible. We can also call, to adjust cacheSize at runtime, and CacheSize means the active node number.
  - set overflow page size. When the size of an item exceeds *maxDataSize*, the extra part in binary form will be transferred to overflow page. And if a page is still not enough, another overflow page will be allocated. Therefore, to save disk space, overflow page should not be too large, and default size is 1024.
- SDB/Skiplist Skiplist is almost the same as btree. For deteminstic skip list,now we only implement 2-skiplist,so it is not need to set degree. Its pageSize and CacheSize are set the same way as btree. And overflow page size is set as the page size.

5. Operation

   For basic operation, Insertion, Deletion, update, get and so on, the client can call the responding method. Note that, SDB/hash don't have *getNext, getPrev, getNearest, getValueBackword, getValueBetween* interface.

```
//commom
bool getValue(const KeyType& key, DataType& data);
bool insertValue(const DataType& data);
bool insertValue(const KeyType& key, const ValueType&
     value);
bool del(const KeyType& key);
bool update( const DataType& data );
bool update(const KeyType& key, const ValueType&
    value);
bool hasKey(const KeyType& key);
void commit();
void flush();
int numItems();

bool search(const KeyType& key, NodeKeyLocn& lonc);
NodeKeyLocn search(const KeyType& key);

//SDB/hash doesn't have ESD_BACKFORD implementation.
bool Seq(NodeKeyLocn& locn, DataType& rec,
    ESeqDirection sdir=ESD_FORWARD);
bool getValueForward(int count, vector<DataType>&
    result);
//
    ============================================================

//for ordered SDB

KeyType getNext(const KeyType& key);
KeyType getPrev(const KeyType& key);
KeyType getNeareas(const KeyType& key);

//if key exists, return itself,otherwise return the
    nearest key bigger than it.
KeyType getNearest(const KeyType& key);
bool getValueForward(const int count, vector<DataType
    >& result,
                        const KeyType& key);
//it starts from begin.
bool getValueForward(int count, vector<DataType>&
    result);
bool getValueBackward(const int count, vector<
    DataType>& result,
                        const KeyType& key);
//it starts from end.
bool getValueBackward(int count, vector<DataType>&
    result);
bool getValueBetween(vector<DataType>& result, const
    KeyType& lowKey,
                const KeyType& highKey);
```

Note that, after insertion or deletion operation, if you can call *commit()* or *flush()* to write the dirty page back to disk. But for efficiency, you can use SequentialDB's internal cache mechanism, when the active node # exceed cacheSize, it will do *flush()* automatically synchronize the modification. And in SequentialDB constructor, it also do *flush()*.

For our Cache mechanism, we maximize the efficiency of Cache to almost keep the tree in memory. However, the size of memory is limited and we should purge some items when memory is full.

## B.3 How to use IndexSDB?

Please see the example: t_index.cc.

1. Define *KeyType*, which is totally the same as SDB/btree. Define *ElementType* Note that if we need prefix query, we should provide method *isPrefix()* .

2. Parameter configuration.

```
typedef unsigned int FieldID;
typedef unsigned int ColID;
typedef string TermID;

typedef unsigned int DocID;

struct KeyType {
  ColID cid;
  FieldID fid;
  TermID tid;

  KeyType() {
  }

  KeyType(ColID c, FieldID f, TermID t) :
   cid(c), fid(f), tid(t) {
 }

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

int compare(const KeyType& other) const {
  if (cid != other.cid)
```

```
    return cid-other.cid;
  else {
    if (fid != other.fid)
       return fid-other.cid;
    else {
       return tid.compare(other.tid);
    }
  }
}

 bool isPrefix(const KeyType& other) const {
  if (typeid(tid).name() != "Ss") {
    return false;
   }

  if (cid != other.cid)
    return false;
  else {
    if (fid != other.fid)
       return false;
    else {
       if (other.tid.substr(0, tid.size() ) == tid) {
          return true;
        else {
          return false;
       }
    }
  }
}

};

//for efficiency, read_image() and write_image() is
    also provided to
//replace default boost::serialization solution.
NS_IZENELIB_AM_BEGIN

namespace util {

template<> inline void read_image< iKeyType<KeyType>
    >(iKeyType<KeyType>& key,
         const DbObjPtr& ptr) {
    memcpy(&key, ptr->getData(), ptr->getSize());
}

template<> inline void write_image< iKeyType<KeyType>
     >(
         const iKeyType<KeyType>& key, DbObjPtr& ptr)
             {
    ptr->setData(&key, sizeof(iKeyType<KeyType>));
}

template<> inline void read_image<std::vector<DocID>
    >(std::vector<DocID>& val,
         const DbObjPtr& ptr) {
    memcpy(&val[0], ptr->getData(), ptr->getSize());
}

template<> inline void write_image<std::vector<DocID>
     >(
         const std::vector<DocID>& val, DbObjPtr& ptr)
              {
    ptr->setData(&val[0], sizeof(DocID)*val.size() );
}

}

NS_IZENELIB_AM_END

IndexSDB<KeyType, DocID> isdb(file);
 //vecSize means how many Element can be store in one
      single node.
size_t vecSize = 100;
degree  = 12;
cacheSize = 100000;

isdb.initialize(vecSize, degree, 500, cacheSize);
```

3. Add or delete an element. If we can make sure the new element will not duplicate with old ones, then *add_nodup* will be more efficient. And *update* can be used for insertion too.

```
bool add(const KeyType& key, const ElementType& item)
    ;
bool add_nodup(const KeyType& key, const ElementType&
     item);
bool del(const KeyType& key);
bool update(const KeyType& key, vector<ElementType>&
    data);
//only write the dirties pages to the disk.
bool commit();
//Beside commit, flush also clear up most of the
    memory.
bool flush();
```

4. Do queries as needed.

```
//if key not exists ,get the key least bigger than it
KeyType getNext(const KeyType& key);
KeyType getPrev(const KeyType& key);
```

```
bool getValue(const KeyType& key, vector<ElementType
    >& result);
bool getValueBetween(const KeyType& lowKey, const
    KeyType& highKey,
                              vector<ElementType>& result);
void getValueIn(const vector<KeyType>& vKey, vector<
    ElementType>& result);
void getValueGreat(const KeyType& key, vector<
    ElementType>& result);
void getValueGreatEqual(const KeyType& key, vector<
    ElementType>& result) ;
void getValueLess(const KeyType& key, vector<
    ElementType>& result);
void getValueLessEqual(const KeyType& key, vector<
    ElementType>& result);
void getValuePrefix(const KeyType& key, vector<
    ElementType>& result);
```

For substring match,we can use another IndexSDB to store all the suffices of the Key(key is string).

```
struct suffix {
  string suf;
  suffix() {
  }
  suffix(string& other) :
   suf(other) {
  }
  template<class Archive> void serialize(Archive & ar
      ,
    const unsigned int version) {
    ar & suf;
  }
  int compare(const suffix& other) const {
    return suf.compare(other.suf);
  }
  bool isPrefix(const suffix& other) const {
    if (other.suf.substr(0, suf.size() ) == suf) {
      return true;
    else {
      return false;
    }
  }
};

string sufFile("suffix.dat");
IndexSDB<suffix, string> sufSDB(sufFile);
sufSDB.initialize(20, degree, 1024, cacheSize);
```

## B.4 How to use SDB-v2 more effectively?

1. Don't use ReadWriteLock for single thread application.
2. Use SDB/btree, SDB/hash, and SDB/tc_hash and tune them well
3. Choose the right parameter
   - SDB/btree(btree_v2)
     - degree When the inserint items are not sequential, Degree from 8 to 16 are the best.
     - pageSize Page Size of 512, 1024, 2048 are recommended. Too big page Size will lead to disk space waste, while too small page Size will lead to overflowing. The best pageSize is

       $$average\ data\ size \times 2 * degree$$

       For many nodes are not full, so some overflow can save disk space.
     - cacheSize The larger*cacheSize*, the better efficiency. But it depends on how much memory is avaible for SDB. *cacheSize* is about

       $$active\ node\ num \times pageSize$$

       . Therefore, we can detemine *cacheSize*from *pageSize*, *degree* and avaible memory.
   - SDB/hash, It'd better choose directorySize and bucketSize depending on the data scale. The rule is to find tradeoff among efficiency and disk space. Note that Long bucket-chain will lead to low efficiency.
     - directorySize
       Too big DirectorySize will lead to disk space waste, while too small directorySize will lead to long bucket-chain and low efficency.
     - bucketSize
       BucketSize should be mulitples of 1024. And too big bucketSize will cause disk space waste, while too small bucketSize will cause long bucket-chain and low efficency.
       The bucket chain should not be too long

       $$average\ bucket\ chain\ length = fileSize/(directorySize * bucketSize)$$

       The best is $1 \sim 2$, and directory size should as large as possible with precodition that bucketSize not too small, at least 1024.
     - cacheSize
       The larger *cacheSize*, the better efficiency. But it depends on how much memory is avaible for SDB. We can determine *cachesize* from how much memory avaible and *bucketSize*.

       $$CacheSize \approx avaible\ memory/bucketSize$$

4. Don't use default *read_image(DataType, DbObj)* and *write_image(DataType, DbObj)* (see, Wrapper.h in source directory) for the KeyType, ValueType and DataType. For SDB/btree_v1, SDB/skiplist, we only need to replace *read_image(DataType, DbObj)* and *write_image(DataType, DbObj)* method for *DataType*. While for SDB/hash, SDB/btree_v2, SDB/tc_hash, SDB/tc_btree, we only need to replace *KeyType* and *ValueType*. See below:

   ```
   typedef int keyType;
   typedef double ValueType;
   typedef izene::am::DataType<KeyType, ValueType>
       DataType;

   namespace izenelib {
   namespace am {
   namespace util {

   template<> inline void read_image<KeyType>(KeyType&
       dat, const DbObjPtr& ptr) {
      memcpy(&dat, ptr->getData(), sizeof(KeyType));
   }

   template<> inline void write_image<KeyType>(const
       KeyType& dat, DbObjPtr& ptr) {
     ptr->setData(&dat, sizeof(dat));
   }

   template<> inline void read_image<ValueType>(
       ValueType& dat, const DbObjPtr& ptr) {
      memcpy(&dat, ptr->getData(), sizeof(ValueType));
   }

   template<> inline void write_image<ValueType>(const
       ValueType& dat, DbObjPtr& ptr) {
     ptr->setData(&dat, sizeof(dat));
   }

   }
   }
   ```

   For user defined DataType, it is highly recommended to replace the default one, including *IndexSDB*.

5. IndexSDB vs SequentialDB

   When key/value pair is like TermID / <DocID >and the size of value(vector) vary much, IndexSDB is better for this case.

6. Rearrange the SDB data file.

   When we delete or update an item, we don't reuse its disk space but leave a page untouched in disk. If there are too many untouched pages, it will low the efficieny. For this case, we'd better dump the SDB data file to a new file when **loadFactor** is low, where loadFactor is used to monitor the pecentage of those untouched pages. For self adjusting data structure like SDB/hash, we can also do self adjusting periodically.

## B.5 How to tune SDB-v2?

We can tune SDB by display its file header infomation after insertion enough number items.

```
SDB.display()
```

### B.5.1 SDB/hash

After SDB/hash call display(), it will print the following message:

```
magic: 398689
bucketSize: 1024
directorySize: 8192
cacheSize: 1000000
numItem: 363529
nBlock: 14531

file size: 14879792 bytes
average items number in bucket: 25.0175
average length of bucket chain: 1.7738
activeNum: 8192
loadFactor: 1
```

- When **average length of bucket chain** is the best within $1 \sim 2$, When it is bigger than 2, we'd better increase *directorySize*. When it is less than 1, we'd better decrease the *directorySize*.
- When **average items number in bucket** is big, we'd better decrease *bucketSize*, when it is small, we'd better increase *bucketSize*.
- Mostly we have to adjust *directorySize* and *bucketSize* both. The right way is to make *directorySize* big and *bucketSize* not too small(at lease 1024).
- CacheSize

  It is recommended as following:

$$CacheSize \approx avaible\ memory\ size/bucketSize$$

### B.5.2 SDB/btree

After SDB/btree call display(), it will print the following message:

```
magic: 398689
maxKeys: 36
pageSize: 1024
cacheSize: 1000000
numItem: 363529
rootPos: 987200
```

**node Pages: 12025**
**overflow Pages: 780**

**file size: 13112384 bytes**
**average items number in a btree ndoe: 30.2311**
**average overflow page for a node: 0.0648649**

- cc-b-btree vs cc-b*-btree

  cc-b-btree works better for ascending sequence, just append the DB dat file with "#". For examep SDB sdb("sdb.dat#"). Otherwisze, cc-b*-btree will be used.

- degree

  For SDB/btree,when the items are not sequential, degree from 8 $\sim$ 16 is recommended. For ascending sequential,big degree 32, 64, 128, can be adopted.

  The tuning point is to try every degree in recommended range.

- pageSize

  When **average overflow page for a node** is within 0.1, It is OK. The disk space efficiency is high. But when it is $\geq$ 0.1, *pagesize* should be increased. When it is 0, it can be decreased to achieve better disk space efficiency.

  For fixed length items, the page size recommended is

  $$sizeof(nodeheader) + 2 * degree * sizeof(item)$$

  where

  $sizeof(nodeheader) = sizeof(byte) + sizeof(size\_t) +$
  $(2 * degree + 1) * sizeof(childr) + sizeof(long) + sizeof(size\_t)$

  ,then no overflow will occur.

- CacheSize

  It is recommended as following:

  $$CacheSize \approx avaible\ memory\ size/pageSize$$

# Index