Design of HugeDB, Draft

Wei Cao

September 17, 2009

Abstract

The SequentialDB data structure is not suitable for large number of random insertions, however many applications rely on such kind of behavior, such as SDBTrie and offline algorithms in MIA. Experiment result and a further analysis show that, the problem of SequentialDB can be due to the inefficient random disk reads and writes during random insertion. So the HugeDB is proposed to replace SequentialDB in such kind of environment. Two major techniques have been raised here to eliminate both random disk reads and random disk writes. One is merge-based incremental growth, which may cause find operations slow down a little if db partitions are unmerged. The other is modification tag method, at the cost of adding a meta field in each record to store modification type. However, the price worth the gain on IO in applications mentioned above.

Contents

1 Document History **1**

2 Introduction **1**

3 Merge-based incremental growth **3**

4 Modification tag **4**

5 Experiment **6**

6 Summary and conclusions **7**

7 Future work **7**

1 Document History

Date	Author	Description
2009-09-16	Wei Cao	An inital version for explaining and reviewing the overall design.

2 Introduction

From now on, for simplity, *sdb* always stands for SequentialDB, while *hdb* is short for HugeDB.

There are two types of sdb, sdb_hash and sdb_btree, keys in sdb_hash are unordered while has order in sdb_btree, so an advantage of btree over hash is that btree could be used for range search. The items in a range of $[x, y]$ can be found by searching for x in the tree, then performing an in-order traversal in the tree from x to y. Although both suffers from rapid performance degradation when db gets large, I focus on sdb_btree, since it's worse and more representive. The following table shows, given plenty of memory (1G), how insertion time increase, they are measured between inserting every 1,000,000 $< int, int >$ records into a sdb_btree continously.

Number of records in sdb_btree	Time spent on inserting next 1,000,000 records
≤ 21,000,000	≤ 1.25 second
22,000,000	**19.9 seconds**
23,000,000	20.2 seconds
...	...
32,000,000	39.0 seconds
33,000,000	41.4 seconds
34,000,000	42.4 seconds
35,000,000	58.5 seconds
36,000,000	1.7 minute
37,000,000	3.3 minutes
38,000,000	3.4 minutes
39,000,000	3.6 minutes
40,000,000	4.1 minutes
41,000,000	4.3 minutes
42,000,000	4.8 minutes
43,000,000	**16.3 minute**s
44,000,000	17.1 minutes

There are two sharp points in the table:

1. The first point is at 22,000,000, before that, all records are kept in memory cache, so insertion speed is rather fast, less than 1.25 second. But at this point, memory cache becomes full, sdb_btree begins continously flushing dirty nodes to disk and loading newly accessed nodes into memory, insertion speed decreases to 19.9 seconds immediately.

2. The second point is at 43,000,000, sdb file's size increase to 2.4G at this point, occupying nearly all 2G physical memory on my machine. The operating system cannot cache the whole file in memory any more, so the OS begins swapping file pages between memory and disk after that point, which causes insertion speed slow down for 4 times immediately.

Above experiment shows, speed of random insertions in sdb_btree degrades too rapidly. Imagine that, 22,000,000 times of insertion can be finished in 1 minute, twice of that, 44,000,000 times of insertions cost nearly 1 hour. Even we assume the speed won't get slow any more, inserting 100,000,000 times will be more than 1 day. I doubt that any user can be satisfied with the performance.

What's wrong with sdb_btree? Is it badly implemented or with poor design? Not really, sdb_btree is just a variant of the classical banlanced multiway B_tree. In classical btree, when a node overflows during in an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Sdb_btree adopts some ideas from B*_tree, splitting is postponed when a node overflows, by "sharing" the node's data with one of its adjacent siblings, the node needs to be split only if the sibling is also full; when that happens, the node splits into two, its data and those of its full sibling are evenly redistrbuted, making each of the three nodes about 2/3 full. This optimization over classical B_tree reduces the number of times new nodes must be created, and thus increase the storage utilization. Another technique sdb_btree used is called "peseundo deletion", Rather than erasing a record from node's page, a user's delete operation just masks the record as invalid and leaves the actual removal to a future insertion operation.

However, sdb_btree is still not enough for large number of random insertions. There are two reasons:

1. In B_tree, every node can be modified during insertion, new record is appended to non-full node and force full node to be splitted, so every block in the sdb file is possible to be modified during random insertions, causing inefficient random seeks and writes. Since the number of dirty nodes flushed out at a time is fixed, the larger sdb file is, sparser those writes will be (here, sparse means how big the gap is between each node's position on disk), disk seeks will cost more time, and time spent on B_tree flush operation will be longer. This is the reason why insertion cost keeps increasing at scale of sdb in experiment.

2. Before each insertion, we must search B_tree to find the right location to insert, if nodes are not in memory, they're loaded from disk file to memory first, this introduces additional random reads. When sdb file is small, operating system could cache the whole file in file cache, and copying a node from OS's file cache to B_tree's node buffer is fast, so developper may not even notice the existing of such activity. But when sdb file grows larger than the size of physical memory, (for example, at 44,000,000, the second sharp point in above table), cost of random reads full play.

OK, since we realize the problem of sdb_btree, how do we prevent random reads and writes during random insertions? Or in other words, shall we convert series of random reads/writes into large sequential read/write? Which would be orders of magnitude faster than former. In section 2, a merge-based incremetal growth method is adopted to archieve this goal. While in section 3, a modification tag method is proposed to eliminate random reads during insertion/updation/deletion.

3 Merge-based incremental growth

Merging technique is well-known for incremental update inverted index. For example, Lucene's indexing bases on two basic algorithms:

1. Make an index for a single document.

2. Merge a set of indices.

Lucene's indexer maintain a stack of segment indices, create index for each incoming document, push new indexes onto the stack, merges them from time to time. Say b is merge factor, M is the number of indexes on the stack, merging algorithm executes in this way:

```
for (size = 1; size < M; size *= b) {
  if (there are b indexes with size docs on top of the stack) {
   pop them off the stack;
   merge them into a single index;
   push the merged index onto the stack;
 } else {
   break;
 }
}
```

Figure 1 shows an example when there are 11 documents and b=3. Merge process executes for 5 times.

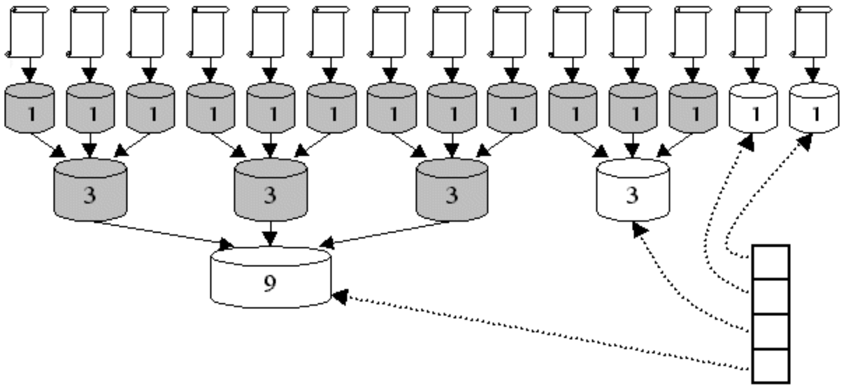

Figure 1: Lucene indexing

Such a merge techinque avoids frequent random modifications to inverted index file, because merge process itself could work in an disk-optimized way just like external sorting algorithm. It reads records sequentially from all input index files, merge records in memory, and output results also in large sequential write. Hence it's not so slow even when inputs are too large to fit in physical memory. So the technique can be adopted to solve our problem, merge sdb_btree on disk from time to time so as to create huge sdb_btree incrementally.

Let us describe it in detail. There are two basic algorithms:

1. Keep an sdb_btree in memory as cache, new records are always inserted into it. Save it as an on-disk sdb_btree when cache gets full, after that clean the cache.

2. Merge on-disk sdb_btree from time to time.

In this way, we can eliminate all random writes, right? Because in algorithm 1 above, storing an in-memory sdb_btree to disk is just to write several handred megabytes memory blocks to disk file for once. And in algorithm 2 merge process only has large sequential reads and writes.

One short coming for this solution is, when in a state there are multiple unmerged sdb_btree structures on disk, we have to search all of them to find value of an given key, hence speed for find operation is affected. In order to apply for those "insert-all-find-all-time" applictions, an optimize() method is provided in hdb, which will force merge all sdb_btree, both in-memory one and on-disk ones, together to be a single sdb.

4 Modification tag

Using merge-based incremental growth method, we can eliminate all random writes. But before inserting a new arrived record into in-memory sdb_btree, we still have to search those on-disk sdb_btree structures, so as to find whether the key exists, which brings large numbers of random reads. Now in this section, the task is to eliminate these random reads.

The premise to slove the problem is, don't try to find whether key exists and its location for new arrival record. The logic is easy to understand, if you search for its location or existance, random read is inevitable.

With this limitation in mind, I proposed a modification tag method. In which, each record is saved together with a tag indicating modification type to show what operation is relevant with the record, e.g, insert, update or delete. So each record in hdb looks like a triple: <key0, value0, INSERT>, <key1, value1, INSERT>, ... , <key0, $value0'$, UPDATE>, <key1, null, DELETE>.

I define four types of tag, INSERT, UPDATE, DELETE to support three basic operations, and an additional one called DELTA for applications that only want to increase orignal value in hdb by some Δ.

- Insert

  For new insertion, say, application developper calls insertValue function:

  ```
  hdb.insertValue(key, value);
  ```

  Hdb just insert the record <key, value, INSERT> into in-memory sdb_btree when no record with the same key exists in in-memory sdb_btree. If such a record exists and tag is INSERT or UPDATE or DELTA, do no modification, or else if tag is DELETE, overwrite it.

- Update

  For update operations like:

  ```
  hdb.updateValue(key, value);
  ```

  Hdb just insert the record <key, value, UPDATE> into in-memory sdb_btree when no record with the same key exists in in-memory sdb_btree. If such a record exists always overwrite it.

- Delete

  ```
  hdb.delete(key);
  ```

  Hdb just insert the record <key, null, DELETE> into in-memory sdb_btree when no record with the same key exists in in-memory sdb_btree. If such a record exists always overwrite it.

- Delta

  A most typical behavior in MIA's offline algorithm is, get the value for a given key, add 1 to the value, then update db.

  ```
  ValueType tmp;
  db.getValue(key, tmp);
  tmp ++;
  db.updateValue(key, tmp);
  ```

  It seems to be a combination of find operation and update operation, which brings random reads. However, fundamentally, here developper doesn't care about exactly what the value is, he just want to increase original value by some Δ (1 in above example).

  So an additional interface called delta() is added to eliminate random reads here, ValueType should have "+" operator, and application developper write code in this way:

  ```
  ValueType delta = 1;
  db.delta(key, delta);
  ```

Hdb just insert the record <key, delta, DELTA> into in-memory sdb_btree when no record with the same key exists in in-memory sdb_btree. If such a record exists,

1. if record is <key, value, INSERT>, overwrite it with <key, value+delta, INSERT> if key doesn't exist in all on-disk sdb_btree, or overwrite it with <key, delta, DELTA>.
2. if record is <key, value, UPDATE>, overwrite it with <key, value+delta, UPDATE>.
3. if record is <key, value, DELTA>, overwrite it with <key, value+delta, DELTA>.
4. if tag is DELETE, overwrite it by the record <key, 1, UPDATE>.

So, it's encouraged to use update rather than insert together with delta, for hdb must search all sdb_btree to see whether key exists so as to decide how to overwrite the record in in-memory sdb_btree.

Just now we talked about how to implement four tyes of operations, they are all computations wholely in memory (except in particular situation user calls insert before delta, but we recommend that, using together with delta, call update instead of insert). As a result, records with the same key may exist between different sdb_btree. So merging algorithm needs small modification to handle them, merging all records with the same key between different sdb_btree. Say there are N records with the same key found in different sdb_btree, stored in an array Records, sorted by sdb_btree's history, here is the code to merge their values:

```
ValueType accumulator;
for( i=0; i<N; i++)
{
  switch( Records[i].tag )
  {
  case INSERT:
  {
    if( accumulator is not set )
      accumulator = Records[i].value;
  }
  case UPDATE:
  {
    accumulator = Records[i].value;
  }
  case DELETE:
  {
    reset accumulator;
  }
  case DELTA:
  {
    accumulator += Records[i].value;
  }
}
```

Find values for a given key from serveral unmerged sdb_btree is implemented in similar mannel. Search all records in separate btrees at first, then merge their values.

5 Experiment

Up to now, I have implemented an initial version of hdb, which supports insert method only. I compare random insertion performance with sdb_btree, whose performance is given in section 1.

By setting merge factor to 4, 100,000,000 times random insertions can be finished within 13 minutes, while ocuppying less than 200M memory.

We can compare the result with sdb_btree, which uses 1G memory, but still takes about one hour to insert 44,000,000 records. (so 100,000,000 times random insertions can be more than one day)

Experiment result shows, eliminating both random reads and random writes boost db's performance visibility for random insertions.

6 Summary and conclusions

In this draft, we check and analyse the reason why sdb_btree has bad performance for random insertions, find it's due to inefficient random reads and writes. Then we propose two techniques: merge-based incremental growth and modification tag to eliminate both of them, although at the cost of much slower find operation when hdb is in unmerged state and store additional 1 byte tag field for each db record. Experiment shows the costs worth gains from efficient IO.

7 Future work

Sdb_btree is just variants of classical banlanced multiway B_tree in textbook, it isn't efficient enough to manage data on massive storage, for example, it doesn't even adopt a B+_tree structure currently. In B+_tree, only leaf node contains data, and internal data have hundreds of child pointers, thus more than 99% pages are leaf pages, making it realistic that all or most internal node cached in memory at most of the time. But in sdb_btree, both internal nodes and leaf nodes contain data, the only difference is leaf nodes do not contain child pointers. I believe find operations can be accerlated by using B+_tree structure.

Besides, large number of materials has been published proposing various versions of B_tree variants, such as write-optimized B_tree, UB-trees and R-trees, understanding how they work and absorbing their advantages into sdb_btree would be of great benifit.