LainDB Documentation

Introduction

LainDB is a very simple key-value store engine. LainDB aims to be fast and flexible at the same time, so it uses modern c++ features, including nullptr and template. Therefore, LainDB needs **c++11 support**. However, Laindb avoids any platform-specific API and can run **on any platform**.

As a key-value store, you are able to store **any kind of data** (you may need to provide a serializer), inside a key (a c-style string with a length limitation). This data can later be retrieved only if we know the exact key used to store it. LainDB provides a minimal interface and it is easy to use but also hard to write unsafe codes.

LainDB's index is implemented by a top-down B+tree. Cache are handled using LRU(Least Recently Used) algorithm.

Quick-Start Guide

example.cpp

```
//include these file to use laindb
#include "../lib/database.hpp"
#include "../lib/utility.hpp"
#include "../lib/optional.hpp"

int main()
{
    laindb::Database<int> db("example"); //open a database
    db.put("sjtu", 1896); //insert a key-value pair
    laindb::Optional<int> res = db.get("sjtu"); //get value, Optional is a type for value
that may exist and may not
    if(res.is_valid()){//check result
        //do something with res.just(), the value of the key
    }
    db.erase("sjtu");//erase a key value pair
    //database will be closed by the destructor
}
```

When compile, link pager.o hashmap.o bptree_index.o & node_store.o.
For more information, see API Reference.

API Reference

Class template: Database

It is the class that represents the whole database. It provides users interfaces to operate the database.

Value: the type of the value.

ValueSerializer: the serializer of the value

Index: the index implementation, laindb only provide B+tree implementation now.

Data: the method to store data, laindb only provide the basic DataStore. It has a parameter for allocator, you can choose between AppendOnlyAllocator and DefaultAllocator.

```
    template <typename Value,
            typename ValueSerializer = DefaultSerializer<Value>,
            typename Index = BptreeIndex,
            typename Data = DataStore<DefaultAllocator> >
    class Database;
```

Constructor

Used to open a database.

Name: name of the data base

Mode: mode to open the file (OPEN: open a database that exists, NEW: create a new database, overwrites the database if it exists, CREATE: try to open first. If fails, create a new database)

```
Database(const std::string & name, FileMode mode = CREATE);
```

Method: get

Fetch the value from the database according to the key. It returns a Optional<Value> because the key-value pair may not exist.

```
Optional<Value> get(const char * key);
```

Method: put

Put a pair of key/value into the database. if the key exists, updates the value.

```
void put(const char * key, const Value & value);
```

Method: erase

Erase a key/value pair from the database according to the key. if the key does not exist, do nothing.

```
void erase(const char * key);
```

class template: Optional

Optional is a type that may contain a value of type T and may not. Like maybe type in some languages.

method: is_valid

test if an optional contains a value.

```
bool is_valid();
```

method: just

get the value (checked)

```
const T & just();
```

Class template: DefaultSerializer

contains methods that converts type T to Bytes and that converts Bytes to type T, the DefaultSerializer is only for **Plain Old Data**, you may need to provide your own serializer, just need to contain the two functions below.

Function: serialize

convert type T to Bytes

```
static Bytes serialize(const T & obj);
```

Function: deserialize

convert Bytes to type T

Side effect: after the call raw will be invalid

```
static T deserialize(Bytes & raw);
```

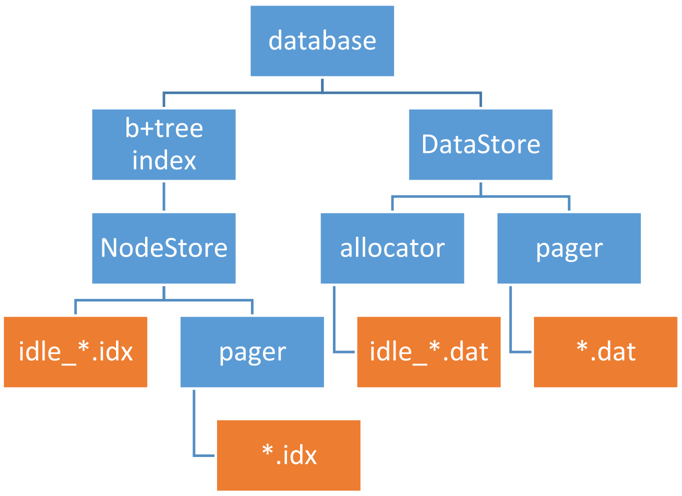
struct: Bytes

This struct is used to represent a string of bytes.

Architecture

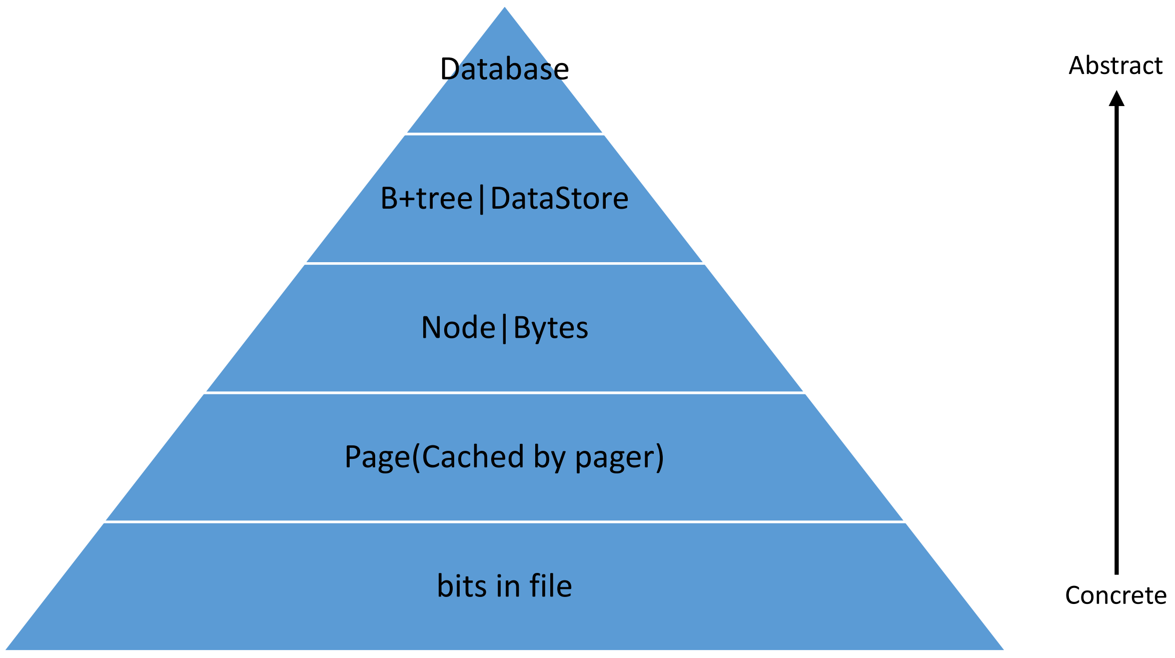
Modules

The LainDB uses the modular design. The relations of the modules are shown in the picture below.



Modules interact with each other using the interface. It is easy to replace them with other modules that implements the interface.

Abstractions with Data



The picture above shows how data are organized. Elements in each layer is construct using the elements in the layer below.

Note:

1. Bytes is a type for serialized value
2. pager is a layer that work as a cache between memory and disk.

Implementation Details

Value in datafile

All values will be serialized to **Bytes** format and then be saved to datafile.

length	content

Bytes

Their address will be allocated by the allocator of the **DataStore.** There are two kind of allocators: AppendOnlyAllocator and DefaultAllocator.

AppendOnlyAllocator just append data at the end of the datafile.

DefaultAllocator uses the best-fit strategy: it will keep deallocated space's information in a linked list, by ascending order of the size. When allocating, it will find the first block that can contain the data. If fails, it will try to merge blocks and try again. Only when the second search fails will the data file grow. DefaultAllocator is a bit slower, but much more economical in space.

Index

LainDB's index is implemented by top-down B+tree. It will try to adjust nodes for operation when going down the tree. It is easier to implement than bottom-up B+tree and avoids recursion.

Node represent:

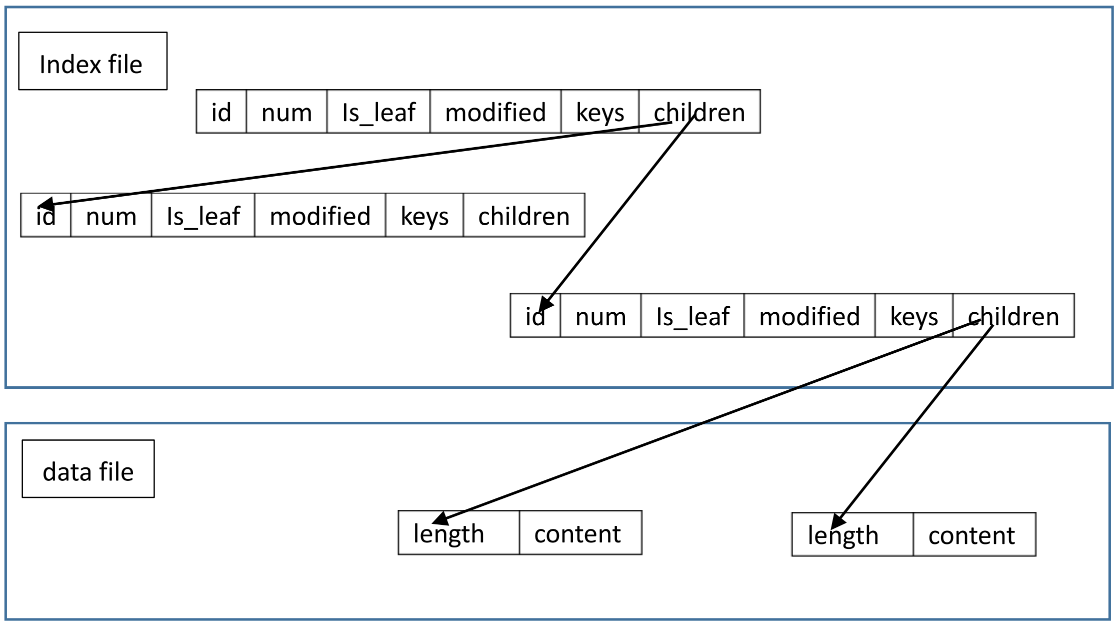
id	num	Is_leaf	modified	keys	children

Note:

Num: number of the keys
Children: in leaves, they are address of data in datafile; in inner nodes, they are IDs of child nodes.

When a b+tree node is stored to the index file, it will be aligned according to the block size of the disk.

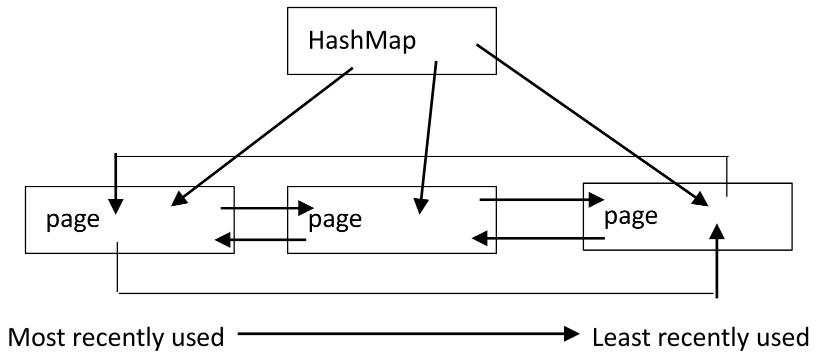
Overview of the store



Pager

Pager is a special layer located in the memory, which functions as a cache for file. Pager manages file in unit of **page**. Page is a segment of file, which size is the same as the block size on disk. When working, some pages will stay in the memory. Pager uses the write-back method to keep consistency i.e. only when modified page is evicted from cache, the page will be written to disk.

The pager uses LRU(Least Recently Used) policy to decide which block will be evicted, when it is full. Pages are organized in a doubly circular linked list. When accessing a page, the page will be move to the head of the list, keeping the most recently used pages at the head of the list and the least at the rear. When a full pager needs to load a new page, the page at the rear of the list will be evicted. Page also uses a HashMap to index pages.



Tests

Correctness Tests

LainDB includes a simple automatic test framework which is developed with macro and template techniques. With it's help, LainDB is well tested by many carefully designed test cases.

Also, LainDB is checked by comparing results with the std::map under random operation sequences.

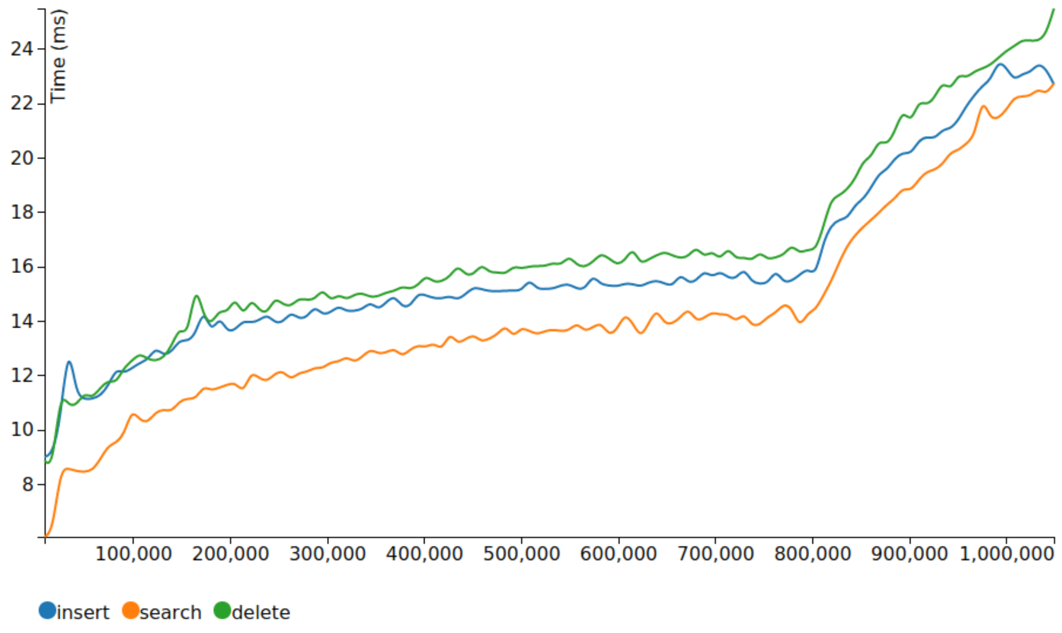
Performance tests

1. **Performance test for insert, search & delete**

   Using the algorithm below:

   1. randomly insert 2^{13} entries
   2. randomly insert 2^{13} entries and record time
   3. randomly search 2^{13} entries and record time
   4. randomly delete 2^{13} entries and record time

   Repeat until the number of entries reaches 2^{20}(1048576). We will get 128 groups of data for time consumed for each operation under different magnitudes of entries. We can use them to draw the picture below.



   It is obvious that the graph can be divided into two parts. The time consumed grows much more fast when the number of entries goes beyond 800,000. This phenomenon show that when there are too many entries, the cache mechanism will become "ineffective". It is because we use random entries which do not have space locality.

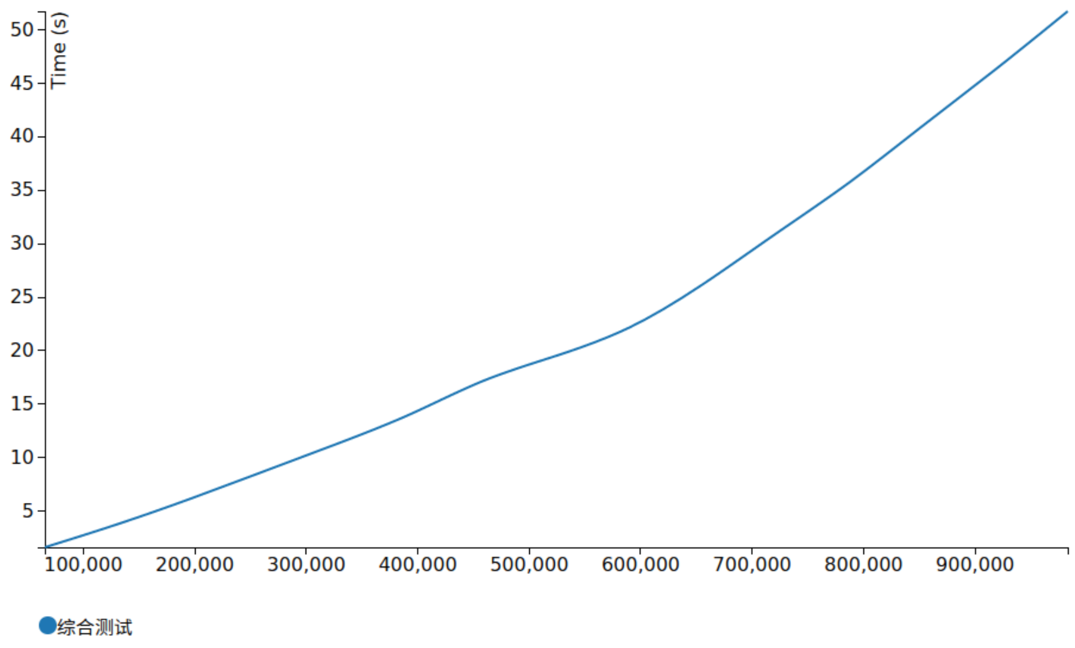
   Both of the two parts of each curve is an approximate log(x) curve, which agrees with the theatrical time complexity of B+tree.

2. **Benchmark**

   Using the method from *Advanced Programming in the UNIX Environment*.

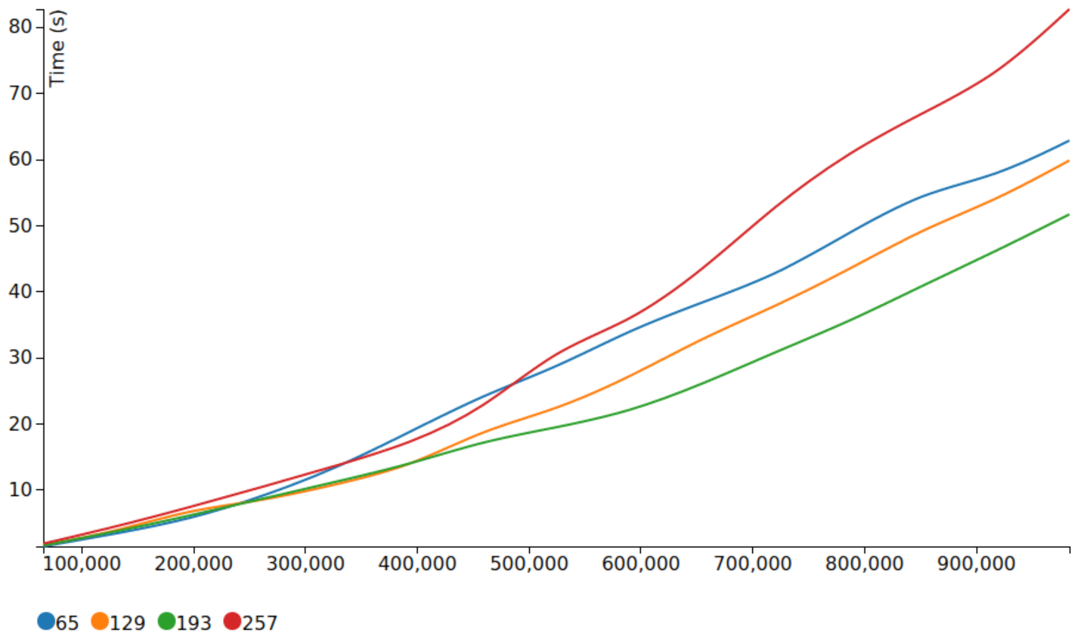
   1. insert NREC entries

2. fetch these entries
3. loop for 5 * NREC times:
   a. randomly fetch an entry
   b. randomly delete an entry, every 37 times
   c. insert an entry and fetch it, every 11 times
   d. randomly replace an entry, every 17 times
4. delete all entries; for each deletion, randomly fetch 10 records.



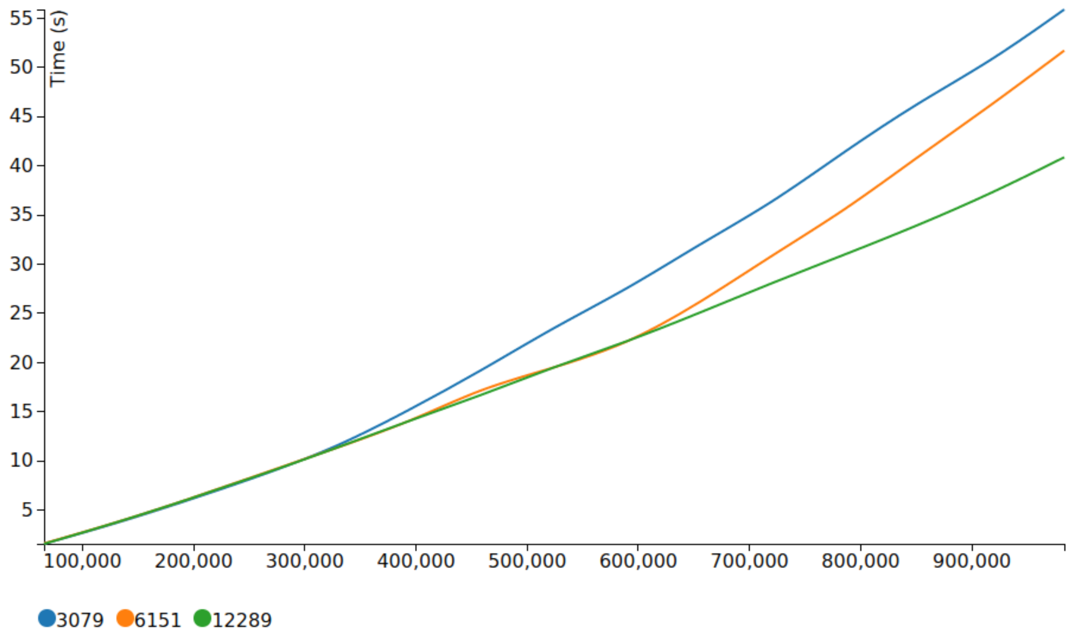
The curve is an approximate nlog(n) curve, which agrees with the theatrical time complexity of B+tree.

3. **Degree of B+tree**



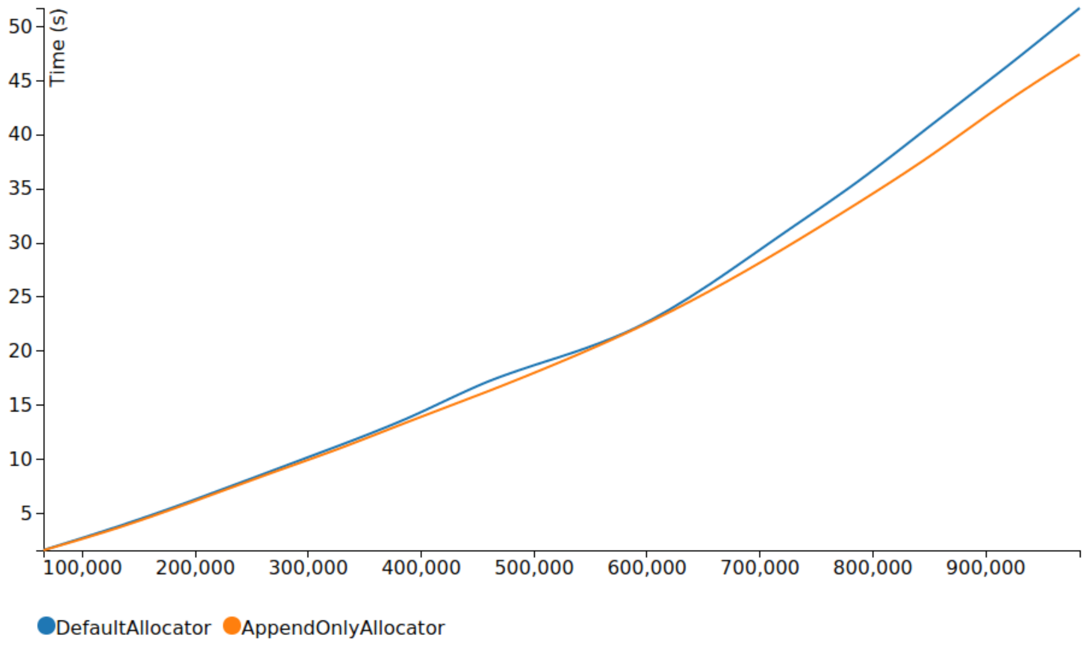
To some extent, large degree will make the database faster, because large degree helps to reduce the height of the tree. However, keeping add degree can make the database slower, because nodes are aligned according to the block size and too large degree may make nodes bigger than a block, which leads to more time to read and write them.

4. **Cache size**



Obviously, with larger cache, database can run faster.

5. **Allocator**



As shown in the graph, the best fit strategy makes the database a bit slower, but makes the data file reduced 24% size (For 1,000,000 benchmark of type int, data file with default allocator is only 7.6MB, while data file with append only allocator is 10MB).