Project for Data Structure

NaiveTweet as a implementation of Social Network

王晟
5120379049

0. Build

0.1 Build project

$make

0.2 Benchmark

$make benchmark

0.3 Clean database

$make cleardb

1. Function

1.1 User management

Users can provide their user name, password and profile to sign up and login using user name and password. After login, the user can change his(her) profile and password. Changing password require providing old password.

1.2 User relation

One can follow other users through user finding function, which can find users by user name, by full name or by profile (birthday and gender). Follow relation is not mutual like that of twitter. User can list people he(she) is following and choose to unfollow.

1.3 Tweet

Tweets are short messages within 140 characters. A user can publish a message and all his(her) followers can see it in their timeline. A user's tweets can also be listed when viewing his(her) profile. Whenever tweets are listed, they can be selected and shared. Shared messages are shown with publisher(sharer) and author.

2. Implementation

2.1 Database

DBMS reads schema from xml file. And creates data structure on disk. For each table, a dat file is created to store data, and several idx files are created for index. The database supports insertion, query, modification and range query. Indexing is implemented using b+tree. Operation on indexed field prefers B+Tree over full scan. The B+Tree records the correspondent file position to the data chunk.

2.2 B+Tree

B+Tree is configured so that it's order will be calculated from block size of storage device and size of provided type, making a node the size of a block on disk. B+Tree has its leaves linked to boost range query. When processing duplicate keys, an overflow block is linked to the leaf for each key, the size of which is also blocksize. Every node in B+Tree is cached to reduce disk operation.

2.3 Cache

In B+Tree, a cached node gets its identity of file position in the idx file. When swapping out, cache control swaps the node which has least file position, causing writing to be linear on disk. According to benchmark, cache size is enough to hold all nodes that are accessed in a short period of time.

3. Correctness

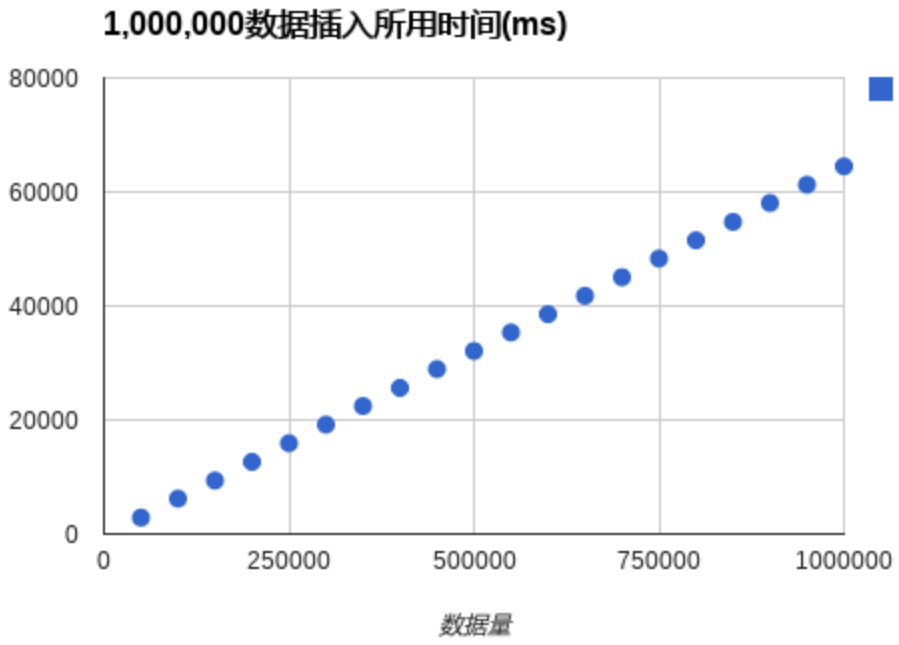
Correctness is tested in benchmark by verifying if insertion data matches queried data. See benchmark section below.

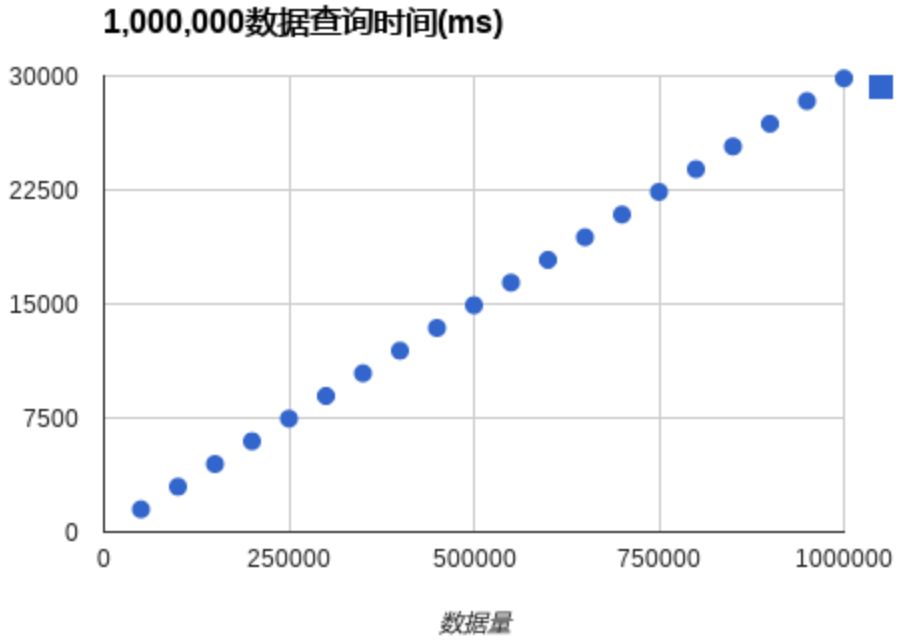
4. Benchmark

DISCLAIMER: The result may differ on differ from hardware. The data only represents relative criterion.

The database is benchmarked by insertion, indexed query and indexed range query 1,000,000 times on my laptop running linux-ck 3.12.13-1 kernel on a i5 3210m cpu. Unindexed query is ten times slower than cached one even on only 100 operation. As it grows O(N^2, it's not benchmarked.
Result shows that time consumption increases approximately at linear rate, which fits expectation of O(NlogN).
Range query of all inserted data is so fast because it doesn't traverse the tree every time but traverse among only leaf node which occurs only once for every data.

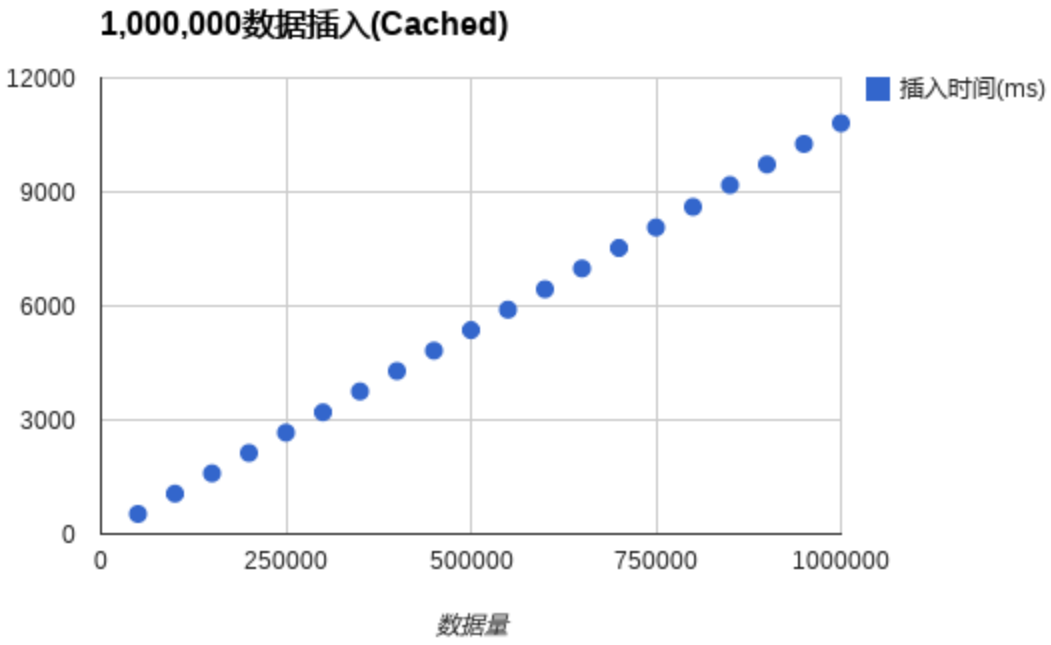
Uncached database operation:

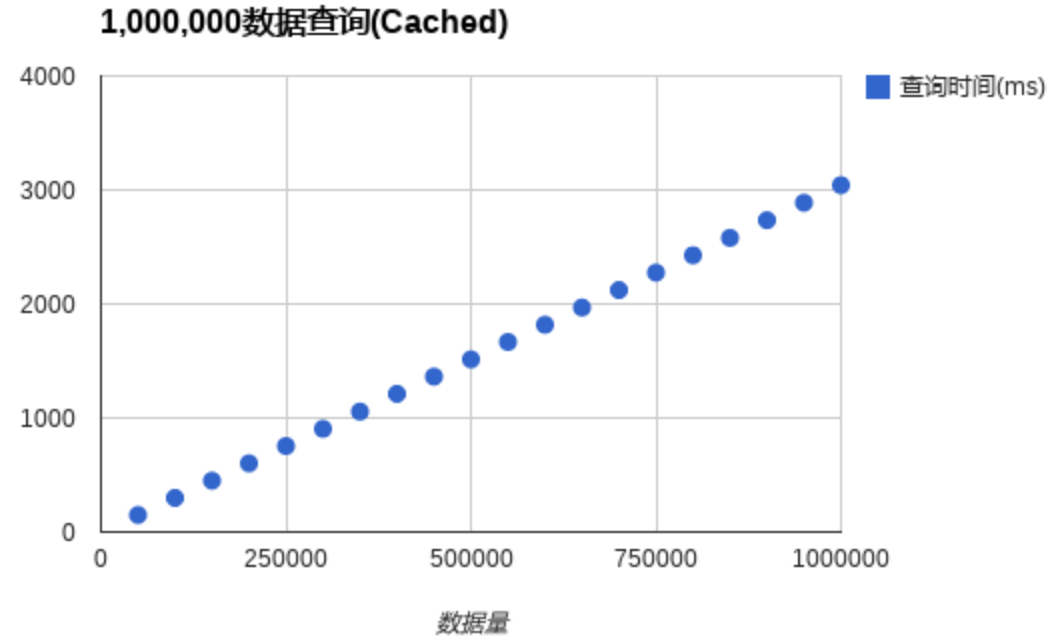




Uncached range query takes 194ms.

Cached database operation:





Cached range query takes 29ms.