# blogbench User Guide

## Summary

Blogbench is a database benchmark to simulate database operations of real-done blog applications. Its main features are the authenticity of the simulation of the web application, easy to use and high flexibility. It can be used to test whether a database is suitable for web application.

## Why we develop blogbench

The reason why we develop blogbench is that the frequently-used benchmarks available have such disadvantages:

* DBT2: DBT2 is a open source implementation of TPS-C, the feature of its application is more of update transactions, less of read transactions. This is different from blog application and lack of flexibility. It can’t be used to only test some type of simple operation, and go against performance optimization.
* Sysbench: sysbench has high flexibility. It includes 9 types of operation. Each operation can be tested separately. This is very useful to performance optimization. But the disadvantages of sysbench are that it only uses a fixed length table and the operation is much different from web application like blog. Such as range query on primary key and sort after query according secondary key rarely occurred in web application. So the result of sysbench is not helpful to proving that database adapts to web application.

## Design objective of blogbench

In view of this，the design objective of blogbench is:

* Simulate real-done blog application. Table schema, feature of data in table, types of operation, and data distributions are designed according our statistic information of blog application.
* High performance. The overhead of benchmark itself is much lower than server’s overhead.
* Flexible. Each type of operation can be tested separately in order to facilitate analyze and optimize performance bottlenecks.

## Table schema and data generation

### Table schema

The test table schema of blogbench is shown bellow (of MySQL):

CREATE TABLE Blog (

ID BIGINT NOT NULL PRIMARY KEY,

UserID BIGINT,

Title VARCHAR(255),

Abstract VARCHAR(2000),

Content MEDIUMTEXT,

AllowView SMALLINT,

PublishTime BIGINT,

AccessCount INT,

CommentCount INT,

KEY IDX\_BLOG\_UID\_PUBTIME(UserID, PublishTime, AllowView)

);

The corresponding table definition of PostgreSQL and Oracle is shown bellow：

* **PostgreSQL**

CREATE TABLE Blog (

ID BIGINT NOT NULL PRIMARY KEY,

UserID BIGINT,

Title VARCHAR(255),

Abstract VARCHAR(2000),

Content TEXT,

AllowView SMALLINT,

PublishTime BIGINT,

AccessCount INT,

CommentCount INT

);

CREATE INDEX IDX\_BLOG\_UID\_PUBTIME ON Blog(UserID, PublishTime, AllowView);

* **Oracle**

CREATE TABLE Blog (

ID NUMBER(20) NOT NULL PRIMARY KEY,

UserID NUMBER(20),

Title VARCHAR2(255),

Abstract VARCHAR2(2000),

Content CLOB,

AllowView NUMBER(5),

PublishTime NUMBER(20),

AccessCount NUMBER(11),

CommentCount NUMBER(11)

);

CREATE INDEX IDX\_BLOG\_UID\_PUBTIME ON Blog(UserID, PublishTime, AllowView);

### Index building process

* For PostgreSQL and Oracle，default behavior is creating primary index when table created, creating secondary index when finish loading data;
* For InnoDB storage engine, default behavior is creating primary index and secondary index when table created;
* For NTSE(Non Transactional Storage Engine developed by NetEase) storage engine, default behavior is create index none when table created, and create primary index and secondary index when finish loading data.

### Data generation

The data generation rules of blogbench is shown bellow:

* ID: from 1 to N，among them N is total number of records;
* UserID: from 1 to N/5,，it means that every user have 5 blogs averagely. UserIDs is as the law of zipf distribution(user-zipf-pct%, user-zipf-res%, user-zipf-part). That means user-zipf-pct% users have user-zipf-res% total blogs, and parted into user-zipf-part levels;
* Title: Title is a string filled with random characters , of which length is as the law of uniform distribution(min-title-size, max-title-size);
* Abstract: Abstract is a string from a real blog article, of which length is as the law of uniform distribution(min-abs-size, max-abs-size);
* Content: Abstract is a string from a real blog article, of which length is as the law of gama distribution(min-cnt-size, max-cnt-size, avg-cnt-size);
* AllowView: random value, 91% is -100, 1% is 100, 8% is 10000;
* PublishTime: current time in format of milliseconds;
* AccessCount: initialized to 0;
* CommentCount: initialized to 0;

You can use many threads to load data concurrently.

## Transactions

### List blog

* **don’t use Memcached**

SELECT

ID, UserID, Title, Abstract, PublishTime, AccessCount, CommentCount

FROM

Blog

WHERE

UserID = ? (UserID)

AND

AllowView <= -100

ORDER BY

PublishTime

DESC LIMIT 10;

* **use Memcached**

// get BlogIDs list from Memcached

id\_list = cache.get("blog:ids:?(UserID)");

if (!id\_list)

//get BlogIDs list from database

id\_list = SELECT ID, UserID FROM Blog WHERE UserID = ?(UserID) AND AllowView <= -100 ORDER BY PublishTime DESC LIMIT 10;

cache.put("blog:ids:?(UserID)", id\_list);

end

cache.multi\_get(id\_list)

// get data of blog columns except content

lightBlogs = SELECT ID, UserID, Title, Abstract, AllowView, PublishTime, AccessCount, CommentCount WHERE (UserID = ?(UserID1) AND ID= ? (id1)) OR (UserID = ?(UserID2) AND ID = ?(id2)) OR ...;

for (each lightBlog in lightBlogs)

cache.put("lblog:?(lightBlog.ID)", lightBlog);

end

INPUT:

* UserID: selected from all UserIDs, using zipf distribution(user-zipf-pct%, user-zipf-res%, user-zipf-part);

### Show blog

* **don’t use Memcached**

SELECT

\*

FROM

Blog

WHERE

ID = ?(BlogID)

AND

UserID = ?(UserID);

* **use Memcached**

lightBlog = cache.get("lblog:?(BlogID)");

if (lightBlog )

content = cache.get("cnt:?"(BlogID));

if (!content)

content = SELECT Content FROM Blog WHERE ID = ?(BlogID) AND UserID = ?(UserID);

cache.put("cnt:?"(BlogID), content);

end

blog = lightBlog + content;

else

content = cache.get("cnt:?(BlogID)");

if (!content) {

blog = SELECT \* FROM Blog WHERE ID = ?(BlogID) AND UserID = ?(UserID);

cache.put("lblog:?(BlogID)", blog.getLightBlog());

cache.put("cnt:?"(BlogID), blog.getContent());

else

lightBlog = SELECT ID, UserID, Title, Abstract, AllowView, PublishTime, AccessCount, CommentCount WHERE UserID = ?(UserID) AND ID = ?(id);

cache.put("lblog:?(BlogID)", lightBlog);

end

INPUT:

* BlogID: selected from all blogs’s BlogIDs, using zipf distribution(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part);
* UserID: UserID of the selected Blog which has the above BlogID.

### Update access count

* **don’t use Memcached**

UPDATE

Blog

SET

AccessCount = AccessCount + 1

WHERE

UserID = ?(UserID)

AND

ID = ?(BlogID);

* **use Memcached**

The process of using Memcached is a bit complex. Blogbench will use a single Memcached server to storage access counts of blogs, called ‘counter\_cache’. The memory allocated to ‘counter\_cache’ is large enough so that the items won’t be replaced very frequently. When update AccessCount property, firstly, query the ‘counter\_cache’ to get the newest access count, and then update it in ‘counter\_cache’. If missed, query the common Memcached server to get the specified blog and put its access count information into ‘counter\_cache’. If missed again in the common Memcached server, query it from database and put the blog into common Memcached server except the content. At the mean time, the <UserID, BlogID, AccessCount> combination is put into local cache that’s made up of several maps. Also there are several background flushing tasks that select some maps to flush access counts into database. If access count of the same blog is updated many times between two flushing processes, the maps only store the newest one. When update access count in common Memcached server or in database, update process will skip it if access count is older in local maps.

There for the process of transaction thread is:

// get the newest access count

blog\_access\_count = counter\_cache.get("blog:?(BlogID)");

if (!blog\_access\_count)

lightBlog = cache.get("lblog:?(BlogID)");

if (!lightBlog)

lightBlog = SELECT ID, UserID, Title, Abstract, AllowView, PublishTime, AccessCount, CommentCount FROM Blog WHERE UserID = ?(UserID) AND ID = ?(BlogID);

cache.put("lblog:?(BlogID)", lightBlog)

end

counter\_cache.put("lblog:?(BlogID)", lightBlog.getAccessCount() + 1);

newAccessCount = lightBlog.getAccessCount() + 1

else

// update access count Memcached server

blog\_access\_count.setAccessCount(blog\_access\_count.getAccessCount() + 1);

counter\_cache.replace("blog:"?(BlogID), blog\_access\_count);

newAccessCount = blog\_access\_count.getAccessCount()

end

// hash to find a map according BlogUserID and put access count into this map

map = all\_maps.get(BlogUserID);

map.set(BlogID, <UserID, BlogID, AccessCount>);

Process of background tasks to flush maps:

while (true)

//get next map to flush

current\_map = next\_map\_to\_flush

// update database and common Memcached server

for (entry in current\_map)

UPDATE Blog SET AccessCount = entry.accessCount WHERE UserID = ?(UserID) AND ID = ?(BlogID) AND AccessCount < entry.accessCount;

lightBlog = cache.get("lblog:?(BlogID)");

if (lightBlog && lightBlog.getAccessCount() < entry.accessCount)

lightBlog.setAccessCount(entry.accessCount)

cache.update("lblog:?(BlogID)", lightBlog)

end

end

// sleep for a short time

sleep(10);

end

INPUT:

* BlogID: selected from all blogs’s BlogIDs, using zipf distribution(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part);
* UserID: UserID of the selected Blog which has the above BlogID.

### Update comment count

* **don’t use Memcached**

UPDATE

Blog

SET

CommentCount = CommentCount + 1

WHERE

UserID = ?(UserID)

AND

ID = ?(BlogID);

* **use Memcached**

UPDATE Blog SET CommentCount = CommentCount + 1 WHERE UserID = ?(UserID) AND ID = ?(BlogID);

lightBlog = cache.get("lblog:"?(BlogID));

if (lightBlog)

lightBlog.setCommentCount(lightBlog.getCommentCount() + 1);

cache.set("blog:"?(BlogID), lightBlog);

end

INPUT:

* BlogID: selected from all blogs’s BlogIDs, using zipf distribution(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part);
* UserID: UserID of the selected Blog which has the above BlogID.

### Show siblings

* **don’t use Memcached**

SELECT

ID, UserID, Title

FROM

Blog

PublishTime < ?(CurrentTime)

AND

AllowView <= -100

AND

UserID = ?(UserID)

ORDER BY

PublishTime

DESC LIMIT 1;

SELECT

ID, UserID, Title

FROM

Blog

PublishTime > ?(CurrentTime)

AND

AllowView <= -100

AND

UserID = ?(UserID)

ORDER BY

PublishTime

ASC LIMIT 1;

* **use Memcached**

id = SELECT ID, UserID FROM Blog PublishTime < ?(CurrentTime) AND AllowView <= -100 AND UserID = ?(UserID) ORDER BY PublishTime DESC LIMIT 1;

cache.get("lblog:?(id)");

if (not exists)

lightBlog = SELECT ID, UserID, Title, Abstract, AllowView, PublishTime, AccessCount, CommentCount FROM Blog WHERE ID = ?(BlogID) AND UserID = ?(UserID);

cache.set("lblog:?(BlogID)", lightBlog);

end

id = SELECT ID, UserID FROM Blog PublishTime > ?(CurrentTime) AND AllowView <= -100 AND UserID = ?(UserID) ORDER BY PublishTime ASC LIMIT 1;

cache.get("lblog:?(id)");

if (not exists)

lightBlog = SELECT ID, UserID, Title, Abstract, AllowView, PublishTime, AccessCount, CommentCount FROM Blog WHERE ID = ?(BlogID) AND UserID = ?(UserID);

cache.set("lblog:?(BlogID)", lightBlog);

end

INPUT:

* UserID: UserID of the Blog that is selected using zipf distribution(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part);
* CurrentTime: PublishTime of the selected blog above;

### Publish blog

* **don’t use Memcached**

INSERT INTO

Blog (ID, UserID, Title, Abstract, Content, AllowView, PublishTime, AccessCount, CommentCount)

VALUES

(?(BlogID), ?(UserID), ?(Title), ?(Abstract), ?(Content), ?(AllowView), ?(PublishTime), 0, 0);

* **use Memcached**

if use memcached, more steps are needed:

//delete blogs list of this user from memcached

cache.del("blog:ids:?(UserID)");

Rules of generation of input parameters is the same as data generation rules.Access probability of the new blog is the same as the probability of selecting a blog using uniform distribution from available blogs, so all blogs are still as the law of zipf distribution.

### Update blog

* **don’t use Memcached**

UPDATE

Blog

SET

PublishTime = ? (CurrentTime), Title= ? (Title), Abstract = ? (Abstract), Content= ? (Content)

WHERE

ID = ? (BlogID)

AND

UserID = ? (UserID);

* **use Memcached**

UPDATE Blog SET PublishTime = ?(CurrentTime), Title = ?(Title), Abstract =? (Abstract), Content = ?(Content) WHERE ID = ?(BlogID) and UserID=?(UserID);

lightBlog = cache.get("lblog:?(BlogID)");

if (lightBlog)

lightBlog.setPublishTime(?(CurrentTime));

...

cache.set("lblog:?(BlogID)", lightBlog);

end

content = cache.get("cnt:?(BlogID)");

if (content)

content.setContent(?(Content));

cache.set("cnt:?(BlogID)", content);

end

INPUT:

* BlogID: selected from all blogs’s BlogIDs, using zipf distribution(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part);
* UserID: UserID of the selected Blog which has the above BlogID.
* CurrentTime, Title, Abstract, Content: the same as the data generation rules;

## Default probability combination of transactions

By default the affairs of the relative proportion are shown below:

|  |  |
| --- | --- |
| transactions | relative proportion |
| list-blogs | 30 |
| show-blog | 60 |
| update-access | 60 |
| update-comment | 10 |
| show-siblings | 60 |
| publish-blog | 10 |
| update-blog | 2 |

## Instructions of blogbench

### Tool packages needed

Most of the scripts files used to collect system information are modified or used directly from Dbt-2 benchmark. It needs these tool packages: libpng, libgd, gnuplot.

### Database character setting

Character set of resource files of blogbench is UTF8. For the same data length to generate, character set of database server should be UTF8.

### Configure before begin blogbench test

Blogbench is not only a java program, but also includes many scripts for collecting system information. In Linux/Unix system, you need to change mod of the script files. This is very simple by executing the ‘configure\_blogbench.sh’ script.

For tests of MySQL or Oracle, you can execute the script file ‘run\_blogbench.sh’. Specified –h option to see more details.

For PostgreSQL, you can write another script according ‘run\_blogbench.sh’.

### Whether need preheating

Normally, blogbench needn’t preheating. The blogbench test is similar to sysbench that test includes two steps: load data and run test. In RUN operation, blogbench will query the corresponding relation of all ID, UserID and PublishTime through this SQL statement:

SELECT ID, UserID, PublishTime FROM table\_name

For these three columns are all in indexes, after this step, index pages will be loaded into memory. So if test duration is enough, blogbench test needn’t extra preheating.

Notice that the RUN operation maybe includes publish-blog transaction and this will change the test table size. You need to load data before begin another test.

### How to use

#### Use jar package directly

Use bellow command to run blogbench:

java com.netease.webbench.blogbench.Main [OPTIONS] [ACTION]

OPTIONS:

* -h/ --host: database host ['127.0.0.1'];
* -P/--port: database port [3306];
* -D/--database: database schema ['test'];
* -u/--user: database user [root];
* -p/--password: database password [''];
* --database-type: database type [mysql];
* --driver-name: name of JDBC driver，optional;
* --jdbc-url: URL of JDBC，optional;
* --table-name: test table name ['Blog'];
* --table-size: number of records to insert into table, only useful for LOAD action [1000000].
* --table-engine: storage engine of test table, only useful for MySQL database type [innodb]
* --table-comment: out of date
* --ntse-create-table-args: none standard create table arguments

['usemms:true; cache\_update:true; cached\_columns:AccessCount'];

* --TRX N: TRX is the name of transaction，N is the relative proportion of a transaction. If N is 0, then this type of transaction is disable;

Possible values of TRX are shown below:

* + list-blogs
  + show-blog
  + update-access
  + update-comment
  + show-siblings
  + publish-blog
  + update-blog
* --blog-zipf-pct N: [5]
* --blog-zipf-res N: [95]
* --blog-zipf-part N: [200]

Zipf distribution used to select blogs. Means that ‘blog-zipf-pct’% blogs hold ‘blog-zipf-res’% access probability, and the hottest blogs are divided into ‘blog-zipf-part’ parts.

* --user-zipf-pct N: [5]
* --user-zipf-res N: [95]
* --user-zipf-part N: [200]

Zipf distribution used to select users. Means that ‘user-zipf-pct’% users hold ‘user-zipf-res’% blogs, and the hottest users are divided into ‘user-zipf-part’ parts.

* --min-title-size N: minimum length of title [10];
* --max-title-size N: maximum length of title, [30];
* --min-abs-size N: minimum length of abstract [10];
* --max-abs-size N: maximum length of abstract [500];
* --min-cnt-size N: minimum length of content [20];
* --max-cnt-size N: maximum length of content [20000];
* --avg-cnt-size N: average length of content [2000];
* --threads N: number of test threads [100];
* --max-tran N: maximum number of transactions [Long.MAX];
* --max-time N: test duration(seconds) [Long.MAX];
* --defer-index: whether defer creating index, only useful for LOAD action. See 2.2 for more details.
* --report-dir: directory for test result [`./result`];
* --collect-sysstat: whether collect system condition automatically [true]
* --print-period: throughput print period(seconds) [60];
* --large-blog: generate extra large blogs [true]. Length of the large blogs is between maxx-cnt-size/2 ~ max-cnt-size, as the law of uniform distribution;
* --use-memcached: if use memcached [false]
* --main-memcached-host: host of memcached server for store blogs [`127.0.0.1`];
* --main-memcached-port: port of memcached server for store blogs [8609];
* --minor-memcached-host: host of memcached server for store blog access count [127.0.0.1];
* --minor-memcached-port：port of memcached server for store blog access count [8608];
* --debug: print debug information, only for development [false];
* --use-two-tables: if divide the blog content into a single table. [false]
* --create-table: if need execute create table statement in load operation. [true]
* --parallel-dml: if use parallel DML(only for Oracle). [false]

ACTION:

* load: load data;
* run: run test;

#### Call from scripts file

Blogbench provides python script to help user configure test cases conveniently. Some common options can be configured through configure file ‘run\_blogbench.cfg’.

* Command line options of run\_blogbench.py:
* -h: print this help message and exit;
* -l / -L / --load: do loading data operation;
* -r / -R / --run: do running test operation;

If both of these two options are specified, they will be executed orderly. If either options is specified, the <operations> configure item in configure file is skipped.

* -c / --configure-file：use this configure file, default is 'run\_blogbench.cfg';
* Options in configure file are show below:
* [environment]
* java\_classpath:class path of java program
* main\_class:main function of blogbench
* [common]
* threads:number of test threads;
* table\_size: number of records to load into test table;
* duration:test duration;
* report\_dir:path of report directory;
* test\_case:name of test case, optional: mix-tran, list-blogs, show-blog, update-access, update-comment, update-blog, publish-blog;
* use\_two\_tables: whether divide the blog content into a single table;
* use\_memcached: whether test with memcached, if it is true, the [memcached] segment is needed.
* operations: type of operations，optional: ‘load’(or ‘l’), ‘run’(or ‘r’). If both operations is needed, it can be divided by ‘,’, such as ‘load,run’. If operation types have been specified through command line, this item is skipped.
* [memcached] (optional)
* main\_memcached\_host: common memcached server host;
* main\_memcached\_port: common memcached server port;
* access\_memcached\_host: access count memcached server host;
* access\_memcached\_port: access count memcached server port;
* [database]
* db\_type: database type，optional: mysql, oracle, postgresql. If it is mysql, the [mysql] segment is needed;
* db\_host: database host;
* db\_port: database port;
* db\_user: database user;
* db\_psw: database password;
* schema\_name: database name;
* [mysql]( optional)
* mysql\_home: installed directory of MySQL;
* mysql\_config: configure file of MySQL;
* mysql\_pid: path of MySQL pid file;
* table\_engine: test table engine;
* auto\_restart: whether restart MySQL server
* [ntse]( optional)
* use\_mms: if test table engine is NTSE, whether use MMS.

## Benchmark result

After finishing blogbench test, these statistic information will be output：

* Throughput of each type of transactions
* Number of each type of transactions finished
* Response time(milliseconds): average、minimum、maximum、90% threshold

### Result files of benchmark

If –report-dir option is not specified, the test result of blogbench is output to ‘./report’ directory. Otherwise is output to specified directory. The test result includes several parts:

* OS runtime information, e.g. CPU, IO etc.
* Database statistic information during test (Only support MySQL currently)
* Result of blogbench includes bellow files：
* blogbench-report.pdf

Blogbench report in chart format;

* blogbench-report.txt

Blogbench report in text format;

* loaddata\_performance.txt

Statistic information of loading data;

* SingleTps-Blogbench.txt

Single TPS of all transactions

* TotalTps-Blogbench.txt

Total TPS of all transactions

* memcached-operation-statistic.txt

This file exists only when –use-memcached option is specified. It includes hit ratio of different types of memcached operation. Acquiescently collect this information every 5 minutes.

## System operation condition monitoring

Blogbench can monitor system operating condition below:

* Condition of OS，such of CPU, IO;
* MySQL configure: result of `show global variables` command (only useful to MySQL);
* MySQL status before and after test: result of `show global status` command (only useful to MySQL);
* Print real time throughput as ASCII chart.

## Others

If Ctrl + C are entered when blogbench test hasn’t finished, then blogbench will catch this signal and save the test result even blogbench hasn’t run for the specified test time.