# blogbench用户手册

## 简述

blogbench为模拟博客真实应用对日志模块所做操作的性能基准测试程序，其主要特征是对应用的模拟真实性高、方便使用、灵活性高。

## 为什么要开发blogbench

开发blogbench的原因是因为现有性能基准测试程序存在以下不足。

* DBT2: DBT2为TPC-C的开源实现，其应用特征为更新事务多，读事务少，与博客应用相差巨大，且不具备灵活性，不能只测试某类简单操作，不利于性能优化；
* sysbench: sysbench灵活性高，包含9类操作，每种操作都可单独测试，对性能优化有很大帮助，但测试只采用一张定长表，操作类型也与博客等Web应用相差很大，如主键上的范围查询、根据辅助索引后的排序等操作在Web应用中很少出现，因此sysbench的结果对验证数据库对Web应用的适应性没多大帮助；

## blogbench的设计目标

鉴于此，blogbench的设计目标为:

* 真实的模拟博客日志应用，包括表模式，表中数据的特征，操作类型，数据分布等都根据博客真实应用的统计结果设定；
* 高效，测试程序本身开销要远小于服务器的开销；
* 灵活，每类操作都可单独进行测试，以方便对NTSE性能瓶颈分析与优化。

## 表模式及数据生成规则

### 表模式

#### 使用一张表

blogbench基准测试使用的测试数据为如下的Blog表(MySQL定义)。

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)

);

对应的PostgreSQL和Oracle的定义如下：

* **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);

#### 使用两张表

当使用两张表时，博客内容字段是被切分到另一张表，两张表的定义分别为：

CREATE TABLE Blog (

ID BIGINT NOT NULL PRIMARY KEY,

UserID BIGINT,

Title VARCHAR(255),

Abstract VARCHAR(2000),

AllowView SMALLINT,

PublishTime BIGINT,

AccessCount INT,

CommentCount INT,

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

);

CREATE TABLE BlogContent (

ID BIGINT NOT NULL PRIMARY KEY,

UserID BIGINT,

Content MEDIUMTEXT,

);

### 索引构建流程

* 对于PostgreSQL和Oracle，默认在建表时建主键索引，数据加载完再建辅助索引；
* 对于InnoDB, 默认建表时建主键索引和辅助索引；
* 对于NTSE(网易开发的非事务型存储引擎)，默认建表不建索引，数据加载完再建主键索引和辅助索引。

### 数据生成规则

数据生成规则如下:

* ID: 从1到N，其中N为总记录数；
* UserID: 从1到N/5，即平均一个用户有5篇日志，使用zipf(user-zipf-pct%, user-zipf-res%, user-zipf-part)分布，即user-zipf-pct%的用户占了user-zipf-res%的概率，分为user-zipf-part级；
* Title: 长度为uniform(min-title-size, max-title-size)字节的字符串，内容为随机字符串；
* Abstract: 长度uniform(min-abs-size, max-abs-size)字节的字符串，内容取自一篇真实的日志文章；
* Content: 长度为min-cnt-size-max-cnt-size字节的字符串，内容取自一篇真实的日志文章。长度使用伽马分布gama(min-cnt-size, max-cnt-size, avg-cnt-size)。
* AllowView: 91%为-100，1%为100，8%为10000;
* PublishTime: 当前时间毫秒数；
* AccessCount: 初始为0；
* CommentCount: 初始为0；

## 操作

### 显示日志列表(list-blogs)

不使用Memcached时

SELECT ID,UserID,Title,Abstract,PublishTime,AccessCount,CommentCount FROM Blog WHERE UserID = ?(UserID) AND AllowView <= -100 ORDER BY PublishTime DESC LIMIT 10;

使用Memcached时

id\_list = cache.get("blog:ids:?(UserID)"); // 使用Memcached获取UserID = ?(UserID) AND AllowView <= -100对应的Blog ID列表

if (!id\_list)

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)

// 从数据库中取出所有cache中不存在的lightBlog

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

输入参数:

* UserID: 从所有UserID中使用zipf(user-zipf-pct%, user-zipf-res%, user-zipf-part)分布选择；

### 显示日志(show-blog)

不使用Memcached时

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

使用Memcached时

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

输入参数:

* BlogID: 从所有BlogID中使用zipf(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part)分布选择；
* UserID: 选定的Blog对应的UserID；

### 更新浏览计数(update-access)

不使用Memcached时

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

使用Memcached时的处理比较复杂。这里将使用一个单独的Memcached系统专用于存储文章的访问计数，称为计数Memcached，counter\_cache。

counter\_cache的内存设置的比较大，使得其中的数据项不会在短期内被换出。在更新AccessCount时，首先查询counter\_cache得到最新的AccessCount，然后修改counter\_cache，如果counter\_cache miss，则从普通Memcached中取得所需信息后插入到counter\_cache，如果普通Memcached又miss，则先从数据库中读取记录然后插入普通Memcached。同时，将<UserID, BlogID, AccessCount>三元组放到由多个Map组成的本地cache中进行去重，同时使用几个后台线程，定期从本地cache中选择一些Map，并刷写到数据库中进行持久化。如果对同一BlogID有多次更新，则在Map中只保存最后的AccessCount。更新Memcached时如果Memcached或数据库中的AccessCount已经比本地cache中的更大，则不更新。

因此事务线程的处理流程如下：

// 取得最新计数

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

// 更新计数Memcached

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

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

newAccessCount = blog\_access\_count.getAccessCount()

end

//根据BlogUserID做哈希找到一个本地Map，将三元组添加到Map中进行去重

map = all\_maps.get(BlogUserID);

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

哈希表处理线程的处理流程如下:

while (true)

//获得下一个待刷新的Map

current\_map = next\_map\_to\_flush

// 更新数据库和普通Memcached

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

// 休眠一小段时间

sleep(10);

end

输入参数:

* BlogID: 从所有BlogID中使用zipf(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part)分布选择；
* UserID: 选定的Blog对应的UserID；

### 更新评论数(update-comment)

不使用Memcached时

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

使用Memcached时

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

输入参数:

* BlogID: 从所有BlogID中使用zipf(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part)分布选择；
* UserID: 选定的Blog对应的UserID；

### 显示前后日志标题(show-siblings)

不使用Memcached时

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;

使用Memcached时

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 (不存在)

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 (不存在)

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

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

end

输入参数:

* UserID: 从所有BlogID中使用zipf(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part)分布选择一个BlogID，得到对应的UserID；
* CurrentTime: 上述被选中的Blog对应的PublishTime；

### 写日志(publish-blog)

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

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

使用Memcached时，另外还要执行

// 设置用户对应的Blog日志列表缓存失效

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

输入参数的生成规则与数据生成规则相同。新生成日志的访问概率为使用均匀分布从已有日志中挑选一篇日志的概率。

### 修改日志(update-blog)

不使用Memcached时

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

使用Memcached时

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

输入参数:

* BlogID: 从所有BlogID中使用zipf(blog-zipf-pct%, blog-zipf-res%, blog-zipf-part)分布选择；
* UserID: 选定的Blog对应的UserID；
* CurrentTime, Title, Abstract, Content生成规则与数据生成规则相同；

## 默认事务概率组合

默认情况下各事务的相对比例如下表所示:

|  |  |
| --- | --- |
| 事务 | 比重 |
| llist-blogs | 30 |
| show-blog | 60 |
| update-access | 60 |
| update-comment | 10 |
| show-siblings | 60 |
| publish-blog | 10 |
| update-blog | 2 |

## 测试程序使用说明

### 需要安装的工具包

目前blogbench的系统信息收集脚本大部分来源于dbt-2测试程序，绘制系统信息图时需要以下的工具包：libpng, libgd, gnuplot。

### 数据库服务器设置

为防止中文乱码，数据库服务器的字符集的设置统一为UTF-8，blogbench启动时会先检查服务器端的字符集设置，如果不是UTF-8，则会抛出异常。(该功能Oracle和PostgreSQL实现还没有测试，所以最好手动检查服务器的字符集设置)

### blogbench测试前的配置

blogbench除了java程序外还带了很多脚本，所以第一次开始运行前，需要先执行configure\_blogbench.sh这个文件，目的是给./scripts目录下的脚本赋权限，因为./scripts目录和里面的文件较多，所以才使用这个方法。如果是测试MySQL或Oracle数据库，可以执行run\_blogbench.sh脚本进行测试，通过-h选项可以了解run\_blogbench.sh更多的帮助信息。如果是其他数据库，建议参照这个脚本另外自己写测试脚本。

### 是否需要预热？

blogbench的测试一般不需要预热，测试采用类似SysBench的方式分为两个阶段: LOAD和RUN，在LOAD的阶段只加载数据，数据加载完需要手动重启服务器；在RUN操作开始前，blogbench会先查询所有的Blog记录的ID， UserID和PublishTime：

SELECT ID, UserID, PublishTime FROM table\_name

通常这个查询会走辅助索引，即这个操作完毕之后索引会被加载到内存中。如果保证测试时间相对表大小来说够长的话，一般不需要预热。

注意RUN操作可能有事务会插入记录，所以RUN测试过程表大小会发生变化，每次重新测试需要重新LOAD数据。

### 使用方法

#### 直接使用jar包

使用以下命令运行测试程序

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

其中OPTIONS包括:

* -h/ --host: 测试所连接的MySQL服务器地址，若不指定则为'127.0.0.1'；
* -P/--port: 测试所连接的MySQL服务器端口，若不指定则为3306；
* -D/--database: 测试所用数据库名，若不指定则为'test'；
* -u/--user: 测试时用于连接数据库服务器的用户名，若不指定则为空；
* -p/--password: 测试时用于连接数据库服务器的密码，若不指定则为空；
* --database-type: 数据库类型，默认为mysql；
* --driver-name: JDBC驱动的名称，可选；
* --jdbc-url: JDBC的URL，可选，当指定的databaseType选项不是mysql时，需要通过这个选项指定JDBC的URL；
* --table-name: 测试所用表名，若不指定则为'Blog'。如果指定了--use-two-tables选项为true,则博客内容表的表名为博客表的表名加'Content'后缀；
* --table-size: 仅当ACTION为load时指定表中包含的记录数，默认为1000000，当ACTION为run时测试程序自动会统计当前表中记录数，指定的--table-size无效；
* --table-engine: 表使用的存储引擎，默认为NTSE，只对MySQL数据库有效；
* --table-comment: 该参数已经废弃，请不要使用；
* --ntse-create-table-args: NTSE非标准表定义信息，只对NTSE有效。默认为'usemms:true;cache\_update:true;cached\_columns:AccessCount'，即启用MMS并对AccessCount字段打开更新缓存功能；
* --TRX N: 总共包含7个参数，TRX为各事务名称，N为相对比重，表示设置某类事务的相对比重，N指定为0表示禁止这类事务；
* --blog-zipf-pct N, --blog-zipf-res N, --blog-zipf-part N: 挑选日志时所用Zipf分布，%--blog-zipf-pct的日志概率占--blog-zipf-res，分为--blog-zipf-part级，默认--blog-zipf-pct为5，--blog-zipf-res为95，--blog-zipf-part为200；
* --user-zipf-pct N, --user-zipf-res N, --user-zipf-part N: 挑选用户时所用Zipf分布，%--user-zipf-pct的日志概率占--user-zipf-res，分为--user-zipf-part级，默认--user-zipf-pct为5，--user-zipf-res为95，--user-zipf-part为200；
* --min-title-size N: Title的最小长度，默认为10；
* --max-title-size N: Title的最大长度，默认为30；
* --min-abs-size N: Abstract的最小长度，默认为10；
* --max-abs-size N: Abstract的最大长度，默认为500；
* --min-cnt-size N: Content的最小长度，默认为20；
* --max-cnt-size N: Content的最大长度，默认为20000；
* --avg-cnt-size N: Content的平均长度，默认为2000；
* --threads N: 使用线程数，默认为100；
* --max-tran N: 测试时最多执行的事务数，默认为Long.MAX；
* --max-time N: 测试时间，默认为Long.MAX，单位为秒；
* --defer-index: 仅当ACTION为load时，表示延后建立索引[true/false], 默认对NTSE为true，对MySQL其他存储引擎、Oracle和PostgreSQL为false；延后建索引意味着数据加载完再建主键索引和辅助索引；非延后建索引意味着建表时建主键索引和辅助索引；
* --report-dir: 指定测试结果文件输出目录，默认为当前目录下的report目录；
* --collect-sysstat: 是否自动收集系统io等信息，默认为true；
* --print-period: 事务吞吐率统计周期,单位为秒，默认为60秒；
* --large-blog：是否生成额外的大记录[true/false]，默认为true。生成的大日志长度在max-cnt-size/2 ~ max-cnt-size之间均匀分布，注意指定该选项后日志平均长度可能比avg-cnt-size稍大。
* --use-memcached:是否启用Memcached[true/false], 默认为false
* --main-memcached-host: 普通记录memcached服务器主机号, 默认为127.0.0.1
* --main-memcached-port: 普通记录memcached服务器端口, 默认为8609
* --minor-memcached-host: 更新计数memcached服务器主机号, 默认为127.0.0.1
* --minor-memcached-port：更新计数memcached服务器端口, 默认为8608
* --clean-mms: 是否在预处理记录时不采用mms（防止mms被heap记录塞满，只对NTSE有效）[true/false], 默认为true
* --debug: 允许输出调试信息（只供开发使用）[true/false], 默认为false
* --use-two-tables：是否将博客内容字段(Content)单独作为一张表，默认为false
* --create-table: 在LOAD操作中是否需要执行建表语句，默认为true
* --parallel-dml: 并行DML(仅供Oracle使用)，默认为false

其中ACTION包括

* load: 装载数据；
* run: 进行测试；

#### 使用脚本启动

blogbench提供方便配置测试用例的python脚本，可通过修改run\_blogbench.cfg配置文件设置测试用例。

* run\_blogbench.py的命令行选项为：
* -h：打印帮助信息；
* -l / -L / --load: 执行加载数据操作；
* -r / -R / --run: 执行测试操作;
* 以上两个选项可以同时指定，同时指定时将顺序执行两个操作；如果任一选项被指定，则在配置文件中的<operations>配置项将不起作用；
* -c / --configure-file：使用指定的配置文件，默认为'run\_blogbench.cfg'；
* 配置文件各选项说明如下：
* [environment]
* java\_classpath：依赖的java类包路径，一般无需修改；
* main\_class：blogbench main函数，一般无需修改；
* [common]
* threads：测试线程；
* table\_size: 测试表记录数；
* duration：测试时长；
* report\_dir：测试结果保存路径；
* test\_case：测试用例名，可选:mix-tran, list-blogs, show-blog, update-access, update-comment, update-blog, publish-blog；
* use\_two\_tables: 是否将博客content字段(大对象)单独存储到一张表；
* use\_memcached: 是否结合memcached进行测试，如果为true，则[memcached]段是必须的；
* operations: 操作类型，可选:load(或l), run(或r)。需要两个操作都执行时可用逗号隔开，如果操作类型已经通过命令行指定，则该项不起作用；
* [memcached] (可选)
* main\_memcached\_host: 普通memcached服务器的主机号；
* main\_memcached\_port: 普通memcached服务器的端口号；
* access\_memcached\_host: 更新计数memcached服务器的主机号；
* access\_memcached\_port: 更新技术memcached服务器的端口号；
* [database]
* db\_type: 数据库类型，可选:mysql, oracle, postgresql。当为mysql时[mysql]段是必须的；
* db\_host: 主机名；
* db\_port: 端口号；
* db\_user: 用户名；
* db\_psw: 密码；
* schema\_name: 测试数据库名称；
* [mysql](可选)
* mysql\_home: MySQL的安装目录；
* mysql\_config: MySQL配置文件路径；
* mysql\_pid: MySQL的pid文件路径；
* table\_engine: 测试表使用的引擎；
* auto\_restart: 是否自动重启MySQL
* [ntse](可选)
* use\_mms: 如果测试ntse引擎，是否使用MMS

## 测试结果

测试程序结束后显示测试结果，包括：

* 事务吞吐率: 每秒执行多少个事务；
* 事务数
* 测试时间
* 响应时间: 平均值、最小值、最大值、90%阈值，单位为毫秒
* 各类在测试时执行的事务的以下信息
* 事务吞吐率
* 事务数
* 响应时间: 平均值、最小值、最大值、90%阈值，单位为毫秒

### 测试结果文件说明

如果没有指定--report-dir目录，则默认的测试结果输出目录为程序所在目录的report目录下，否则为指定的--report-dir目录。结果目录下的测试结果包含两部分：

* 系统cpu，io信息等(如果设置了--collect-sysstat为off，则blogbench不会主动去调用收集系统信息的脚本)
* 数据库的状态与变量信息(只对MySQL会自动收集)
* blogbench的测试结果，包含以下几个文件：
* blogbench-report.pdf

最主要的测试结果文件，包含事务吞吐率，事务响应时间分布，如果指定了使用memcached，还会有不同事务各种memcached操作的命中率统计；

* blogbench-report.txt

在blogbench-report.pdf文件中已经包含了这个结果，这里写入txt文件是为了防止不能生成pdf文件时失去测试结果。

* loaddata\_performance.txt

load数据的统计信息，包括建表，insert记录，建索引的时间；

* SingleTps-Blogbench.txt

七种事务在测试过程中每个周期的tps

* TotalTps-Blogbench.txt

七种事务在测试过程中每个周期的的总的tps

* memcached-operation-statistic.txt

指定了使用memcached时才有这个统计结果，包含测试过程中不同事务各种memcached操作的命中率，默认5分钟统计一次。

## 系统运行状态监控

blogbench可以监控并记录以下系统运行状态:

* 操作系统状态，如cpu，io等；
* MySQL配置: show global variables的结果(只对MySQL有效)
* 测试前后MySQL状态: show global status的结果(只对MySQL有效)
* 测试期间每隔5分钟输出一次NTSE information\_schema表信息(只对MySQL&&NTSE有效)
* 以ASCII图的形式显示实时吞吐率

## 其他功能

如果测试过程还没达到指定的--max-time时间，又想结束测试，在Linux下可以按Ctrl + C，blogbench会捕获这个信号并将测试结果保存之后退出。这时得到的测试结果就是程序启动到被终止过程中的测试结果。