NAME

pgbench − run a benchmark test on PostgreSQL

SYNOPSIS

pgbench −i [*option*...] [*dbname*]

pgbench [*option*...] [*dbname*]

DESCRIPTION

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC−B, involving five **SELECT**, **UPDATE**, and **INSERT** commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

Typical output from pgbench looks like:

```
transaction type: <builtin: TPC−B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
latency average = 11.013 ms
latency stddev = 7.351 ms
initial connection time = 45.758 ms
tps = 896.967014 (without initial connection time)
```

The first seven lines report some of the most important parameter settings. The sixth line reports the maximum number of tries for transactions with serialization or deadlock errors (see Failures and Serialization/Deadlock Retries for more information). The eighth line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions per client); these will be equal unless the run failed before completion or some SQL command(s) failed. (In **−T** mode, only the actual number of transactions is printed.) The next line reports the number of failed transactions due to serialization or deadlock errors (see Failures and Serialization/Deadlock Retries for more information). The last line reports the number of transactions per second.

The default TPC−B−like transaction test requires specific tables to be set up beforehand. pgbench should be invoked with the **−i** (initialize) option to create and populate these tables. (When you are testing a custom script, you don't need this step, but will instead need to do whatever setup your test needs.) Initialization looks like:

```
pgbench −i [ other−options ] dbname
```

where *dbname* is the name of the already−created database to test in. (You may also need **−h**, **−p**, and/or **−U** options to specify how to connect to the database server.)

Caution

pgbench −i creates four tables pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers, destroying any existing tables of these names. Be very careful to use another database if you have tables having these names!

At the default “scale factor” of 1, the tables initially contain this many rows:

```
table                # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

You can (and, for most purposes, probably should) increase the number of rows by using the **−s** (scale factor) option. The **−F** (fillfactor) option might also be used at this point.

Once you have done the necessary setup, you can run your benchmark with a command that doesn't include **−i**, that is

```
pgbench [ options ] dbname
```

In nearly all cases, you'll need some options to make a useful test. The most important options are **−c** (number of clients), **−t** (number of transactions), **−T** (time limit), and **−f** (specify a custom script file). See below for a full list.

OPTIONS

The following is divided into three subsections. Different options are used during database initialization and while running benchmarks, but some options are useful in both cases.

Initialization Options

pgbench accepts the following command−line initialization arguments:

dbname

Specifies the name of the database to test in. If this is not specified, the environment variable **PGDATABASE** is used. If that is not set, the user name specified for the connection is used.

−i
−−initialize

Required to invoke initialization mode.

−I *init_steps*
−−init−steps=*init_steps*

Perform just a selected set of the normal initialization steps. *init_steps* specifies the initialization steps to be performed, using one character per step. Each step is invoked in the specified order. The default is dtgvp. The available steps are:

d (Drop)

Drop any existing pgbench tables.

t (create Tables)

Create the tables used by the standard pgbench scenario, namely pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers.

g or G (Generate data, client−side or server−side)

Generate data and load it into the standard tables, replacing any data already present.

With g (client−side data generation), data is generated in **pgbench** client and then sent to the server. This uses the client/server bandwidth extensively through a **COPY**. **pgbench** uses the FREEZE option with version 14 or later of PostgreSQL to speed up subsequent **VACUUM**, unless partitions are enabled. Using g causes logging to print one message every 100,000 rows while generating data for the pgbench_accounts table.

With G (server−side data generation), only small queries are sent from the **pgbench** client and then data is actually generated in the server. No significant bandwidth is required for this variant, but the server will do more work. Using G causes logging not to print any progress message while generating data.

The default initialization behavior uses client−side data generation (equivalent to g).

v (Vacuum)
: Invoke **VACUUM** on the standard tables.

p (create Primary keys)
: Create primary key indexes on the standard tables.

f (create Foreign keys)
: Create foreign key constraints between the standard tables. (Note that this step is not performed by default.)

−F *fillfactor*
−−fillfactor=*fillfactor*
: Create the pgbench_accounts, pgbench_tellers and pgbench_branches tables with the given fillfactor. Default is 100.

−n
−−no−vacuum
: Perform no vacuuming during initialization. (This option suppresses the v initialization step, even if it was specified in **−I**.)

−q
−−quiet
: Switch logging to quiet mode, producing only one progress message per 5 seconds. The default logging prints one message each 100,000 rows, which often outputs many lines per second (especially on good hardware).

  This setting has no effect if G is specified in **−I**.

−s *scale_factor*
−−scale=*scale_factor*
: Multiply the number of rows generated by the scale factor. For example, −s 100 will create 10,000,000 rows in the pgbench_accounts table. Default is 1. When the scale is 20,000 or larger, the columns used to hold account identifiers (aid columns) will switch to using larger integers (bigint), in order to be big enough to hold the range of account identifiers.

−−foreign−keys
: Create foreign key constraints between the standard tables. (This option adds the f step to the initialization step sequence, if it is not already present.)

−−index−tablespace=*index_tablespace*
: Create indexes in the specified tablespace, rather than the default tablespace.

−−partition−method=*NAME*
: Create a partitioned pgbench_accounts table with *NAME* method. Expected values are range or hash. This option requires that **−−partitions** is set to non−zero. If unspecified, default is range.

−−partitions=*NUM*
: Create a partitioned pgbench_accounts table with *NUM* partitions of nearly equal size for the scaled number of accounts. Default is 0, meaning no partitioning.

−−tablespace=*tablespace*
: Create tables in the specified tablespace, rather than the default tablespace.

−−unlogged−tables
: Create all tables as unlogged tables, rather than permanent tables.

Benchmarking Options

pgbench accepts the following command−line benchmarking arguments:

−b *scriptname[@weight]*
−−builtin=*scriptname[@weight]*

Add the specified built–in script to the list of scripts to be executed. Available built–in scripts are: tpcb–like, simple–update and select–only. Unambiguous prefixes of built–in names are accepted. With the special name list, show the list of built–in scripts and exit immediately.

Optionally, write an integer weight after @ to adjust the probability of selecting this script versus other ones. The default weight is 1. See below for details.

–c *clients*
––client=*clients*
Number of clients simulated, that is, number of concurrent database sessions. Default is 1.

–C
––connect
Establish a new connection for each transaction, rather than doing it just once per client session. This is useful to measure the connection overhead.

–d
––debug
Print debugging output.

–D *varname*=*value*
––define=*varname*=*value*
Define a variable for use by a custom script (see below). Multiple **–D** options are allowed.

–f *filename[@weight]*
––file=*filename[@weight]*
Add a transaction script read from *filename* to the list of scripts to be executed.

Optionally, write an integer weight after @ to adjust the probability of selecting this script versus other ones. The default weight is 1. (To use a script file name that includes an @ character, append a weight so that there is no ambiguity, for example filen@me@1.) See below for details.

–j *threads*
––jobs=*threads*
Number of worker threads within pgbench. Using more than one thread can be helpful on multi–CPU machines. Clients are distributed as evenly as possible among available threads. Default is 1.

–l
––log
Write information about each transaction to a log file. See below for details.

–L *limit*
––latency–limit=*limit*
Transactions that last more than *limit* milliseconds are counted and reported separately, as late.

When throttling is used (**––rate=...**), transactions that lag behind schedule by more than *limit* ms, and thus have no hope of meeting the latency limit, are not sent to the server at all. They are counted and reported separately as skipped.

When the **––max–tries** option is used, a transaction which fails due to a serialization anomaly or from a deadlock will not be retried if the total time of all its tries is greater than *limit* ms. To limit only the time of tries and not their number, use ––max–tries=0. By default, the option **––max–tries** is set to 1 and transactions with serialization/deadlock errors are not retried. See Failures and Serialization/Deadlock Retries for more information about retrying such transactions.

–M *querymode*
––protocol=*querymode*
Protocol to use for submitting queries to the server:

- simple: use simple query protocol.
- extended: use extended query protocol.
- prepared: use extended query protocol with prepared statements.

In the prepared mode, pgbench reuses the parse analysis result starting from the second query iteration, so pgbench runs faster than in other modes.

The default is simple query protocol. (See Chapter 55 for more information.)

−n
−−no−vacuum
Perform no vacuuming before running the test. This option is *necessary* if you are running a custom test scenario that does not include the standard tables pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers.

−N
−−skip−some−updates
Run built−in simple−update script. Shorthand for **−b simple−update**.

−P *sec*
−−progress=*sec*
Show progress report every *sec* seconds. The report includes the time since the beginning of the run, the TPS since the last report, and the transaction latency average, standard deviation, and the number of failed transactions since the last report. Under throttling (**−R**), the latency is computed with respect to the transaction scheduled start time, not the actual transaction beginning time, thus it also includes the average schedule lag time. When **−−max−tries** is used to enable transaction retries after serialization/deadlock errors, the report includes the number of retried transactions and the sum of all retries.

−r
−−report−per−command
Report the following statistics for each command after the benchmark finishes: the average per−statement latency (execution time from the perspective of the client), the number of failures, and the number of retries after serialization or deadlock errors in this command. The report displays retry statistics only if the **−−max−tries** option is not equal to 1.

−R *rate*
−−rate=*rate*
Execute transactions targeting the specified rate instead of running as fast as possible (the default). The rate is given in transactions per second. If the targeted rate is above the maximum possible rate, the rate limit won't impact the results.

The rate is targeted by starting transactions along a Poisson−distributed schedule time line. The expected start time schedule moves forward based on when the client first started, not when the previous transaction ended. That approach means that when transactions go past their original scheduled end time, it is possible for later ones to catch up again.

When throttling is active, the transaction latency reported at the end of the run is calculated from the scheduled start times, so it includes the time each transaction had to wait for the previous transaction to finish. The wait time is called the schedule lag time, and its average and maximum are also reported separately. The transaction latency with respect to the actual transaction start time, i.e., the time spent executing the transaction in the database, can be computed by subtracting the schedule lag time from the reported latency.

If **−−latency−limit** is used together with **−−rate**, a transaction can lag behind so much that it is already over the latency limit when the previous transaction ends, because the latency is calculated from the scheduled start time. Such transactions are not sent to the server, but are skipped altogether and counted separately.

A high schedule lag time is an indication that the system cannot process transactions at the specified rate, with the chosen number of clients and threads. When the average transaction execution time is longer than the scheduled interval between each transaction, each successive transaction will fall further behind, and the schedule lag time will keep increasing the longer the test run is. When that happens, you will have to reduce the specified transaction rate.

−s *scale_factor*
−−scale=*scale_factor*
Report the specified scale factor in pgbench's output. With the built−in tests, this is not necessary; the correct scale factor will be detected by counting the number of rows in the pgbench_branches table. However, when testing only custom benchmarks (**−f** option), the scale factor will be reported as 1 unless this option is used.

−S
−−select−only
Run built−in select−only script. Shorthand for **−b select−only**.

−t *transactions*
−−transactions=*transactions*
Number of transactions each client runs. Default is 10.

−T *seconds*
−−time=*seconds*
Run the test for this many seconds, rather than a fixed number of transactions per client. **−t** and **−T** are mutually exclusive.

−v
−−vacuum−all
Vacuum all four standard tables before running the test. With neither **−n** nor **−v**, pgbench will vacuum the pgbench_tellers and pgbench_branches tables, and will truncate pgbench_history.

−−aggregate−interval=*seconds*
Length of aggregation interval (in seconds). May be used only with **−l** option. With this option, the log contains per−interval summary data, as described below.

−−failures−detailed
Report failures in per−transaction and aggregation logs, as well as in the main and per−script reports, grouped by the following types:

- serialization failures;
- deadlock failures;

See Failures and Serialization/Deadlock Retries for more information.

−−log−prefix=*prefix*
Set the filename prefix for the log files created by **−−log**. The default is pgbench_log.

−−max−tries=*number_of_tries*
Enable retries for transactions with serialization/deadlock errors and set the maximum number of these tries. This option can be combined with the **−−latency−limit** option which limits the total time of all transaction tries; moreover, you cannot use an unlimited number of tries (−−max−tries=0) without **−−latency−limit** or **−−time**. The default value is 1 and transactions with serialization/deadlock errors are not retried. See Failures and Serialization/Deadlock Retries for more information about retrying such transactions.

−−progress−timestamp
When showing progress (option **−P**), use a timestamp (Unix epoch) instead of the number of seconds since the beginning of the run. The unit is in seconds, with millisecond precision after the dot. This helps compare logs generated by various tools.

−−random−seed=*seed*
: Set random generator seed. Seeds the system random number generator, which then produces a sequence of initial generator states, one for each thread. Values for *seed* may be: time (the default, the seed is based on the current time), rand (use a strong random source, failing if none is available), or an unsigned decimal integer value. The random generator is invoked explicitly from a pgbench script (random... functions) or implicitly (for instance option **−−rate** uses it to schedule transactions). When explicitly set, the value used for seeding is shown on the terminal. Any value allowed for *seed* may also be provided through the environment variable PGBENCH_RANDOM_SEED. To ensure that the provided seed impacts all possible uses, put this option first or use the environment variable.

  Setting the seed explicitly allows to reproduce a **pgbench** run exactly, as far as random numbers are concerned. As the random state is managed per thread, this means the exact same **pgbench** run for an identical invocation if there is one client per thread and there are no external or data dependencies. From a statistical viewpoint reproducing runs exactly is a bad idea because it can hide the performance variability or improve performance unduly, e.g., by hitting the same pages as a previous run. However, it may also be of great help for debugging, for instance re−running a tricky case which leads to an error. Use wisely.

−−sampling−rate=*rate*
: Sampling rate, used when writing data into the log, to reduce the amount of log generated. If this option is given, only the specified fraction of transactions are logged. 1.0 means all transactions will be logged, 0.05 means only 5% of the transactions will be logged.

  Remember to take the sampling rate into account when processing the log file. For example, when computing TPS values, you need to multiply the numbers accordingly (e.g., with 0.01 sample rate, you'll only get 1/100 of the actual TPS).

−−show−script=*scriptname*
: Show the actual code of builtin script *scriptname* on stderr, and exit immediately.

−−verbose−errors
: Print messages about all errors and failures (errors without retrying) including which limit for retries was exceeded and how far it was exceeded for the serialization/deadlock failures. (Note that in this case the output can be significantly increased.). See Failures and Serialization/Deadlock Retries for more information.

Common Options

pgbench also accepts the following common command−line arguments for connection parameters:

−h *hostname*
−−host=*hostname*
: The database server's host name

−p *port*
−−port=*port*
: The database server's port number

−U *login*
−−username=*login*
: The user name to connect as

−V
−−version
: Print the pgbench version and exit.

−?
−−help
: Show help about pgbench command line arguments, and exit.

EXIT STATUS

A successful run will exit with status 0. Exit status 1 indicates static problems such as invalid command−line options or internal errors which are supposed to never occur. Early errors that occur when starting benchmark such as initial connection failures also exit with status 1. Errors during the run such as database errors or problems in the script will result in exit status 2. In the latter case, pgbench will print partial results.

ENVIRONMENT

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters.

This utility, like most other PostgreSQL utilities, uses the environment variables supported by libpq (see Section 34.15).

The environment variable **PG_COLOR** specifies whether to use color in diagnostic messages. Possible values are always, auto and never.

NOTES

What Is the “Transaction” Actually Performed in pgbench?

pgbench executes test scripts chosen randomly from a specified list. The scripts may include built−in scripts specified with **−b** and user−provided scripts specified with **−f**. Each script may be given a relative weight specified after an @ so as to change its selection probability. The default weight is 1. Scripts with a weight of 0 are ignored.

The default built−in transaction script (also invoked with **−b tpcb−like**) issues seven commands per transaction over randomly chosen aid, tid, bid and delta. The scenario is inspired by the TPC−B benchmark, but is not actually TPC−B, hence the name.

1. BEGIN;
2. UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
4. UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
7. END;

If you select the simple−update built−in (also **−N**), steps 4 and 5 aren't included in the transaction. This will avoid update contention on these tables, but it makes the test case even less like TPC−B.

If you select the select−only built−in (also **−S**), only the **SELECT** is issued.

Custom Scripts

pgbench has support for running custom benchmark scenarios by replacing the default transaction script (described above) with a transaction script read from a file (**−f** option). In this case a “transaction” counts as one execution of a script file.

A script file contains one or more SQL commands terminated by semicolons. Empty lines and lines beginning with −− are ignored. Script files can also contain “meta commands”, which are interpreted by pgbench itself, as described below.

Note

Before PostgreSQL 9.6, SQL commands in script files were terminated by newlines, and so they could not be continued across lines. Now a semicolon is *required* to separate consecutive SQL commands (though an SQL command does not need one if it is followed by a meta command). If you need to create a script file that works with both old and new versions of pgbench, be sure to write each SQL command on a single line ending with a semicolon.

It is assumed that pgbench scripts do not contain incomplete blocks of SQL transactions. If at runtime the client reaches the end of the script without completing the last transaction block, it will be aborted.

There is a simple variable−substitution facility for script files. Variable names must consist of letters (including non−Latin letters), digits, and underscores, with the first character not being a digit. Variables can be set by the command−line **−D** option, explained above, or by the meta commands explained below. In addition to any variables preset by **−D** command−line options, there are a few variables that are preset automatically, listed in Table 288. A value specified for these variables using **−D** takes precedence over the automatic presets. Once set, a variable's value can be inserted into an SQL command by writing :*variablename*. When running more than one client session, each session has its own set of variables. pgbench supports up to 255 variable uses in one statement.

Table 288. pgbench Automatic Variables

Variable	**Description**
client_id	unique number identifying the client session (starts from zero)
default_seed	seed used in hash and pseudorandom permutation functions by default
random_seed	random generator seed (unless overwritten with **−D**)
scale	current scale factor

Script file meta commands begin with a backslash (\) and normally extend to the end of the line, although they can be continued to additional lines by writing backslash−return. Arguments to a meta command are separated by white space. These meta commands are supported:

\gset [*prefix*] \aset [*prefix*]

These commands may be used to end SQL queries, taking the place of the terminating semicolon (;).

When the \gset command is used, the preceding SQL query is expected to return one row, the columns of which are stored into variables named after column names, and prefixed with *prefix* if provided.

When the \aset command is used, all combined SQL queries (separated by \;) have their columns stored into variables named after column names, and prefixed with *prefix* if provided. If a query returns no row, no assignment is made and the variable can be tested for existence to detect this. If a query returns more than one row, the last value is kept.

\gset and \aset cannot be used in pipeline mode, since the query results are not yet available by the time the commands would need them.

The following example puts the final account balance from the first query into variable *abalance*, and fills variables *p_two* and *p_three* with integers from the third query. The result of the second query is discarded. The result of the two last combined queries are stored in variables *four* and *five*.

```
UPDATE pgbench_accounts
  SET abalance = abalance + :delta
  WHERE aid = :aid
  RETURNING abalance \gset
−− compound of two queries
SELECT 1 \;
SELECT 2 AS two, 3 AS three \gset p_
SELECT 4 AS four \; SELECT 5 AS five \aset
```

\if *expression*
\elif *expression*
\else
\endif

This group of commands implements nestable conditional blocks, similarly to psql's \if *expression*. Conditional expressions are identical to those with \set, with non−zero values interpreted as true.

\set *varname expression*

Sets variable *varname* to a value calculated from *expression*. The expression may contain the NULL constant, Boolean constants TRUE and FALSE, integer constants such as 5432, double constants such as 3.14159, references to variables :*variablename*, operators with their usual SQL precedence and associativity, function calls, SQL CASE generic conditional expressions and parentheses.

Functions and most operators return NULL on NULL input.

For conditional purposes, non zero numerical values are TRUE, zero numerical values and NULL are FALSE.

Too large or small integer and double constants, as well as integer arithmetic operators (+, −, * and /) raise errors on overflows.

When no final ELSE clause is provided to a CASE, the default value is NULL.

Examples:

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
         (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

\sleep *number* [us | ms | s]

Causes script execution to sleep for the specified duration in microseconds (us), milliseconds (ms) or seconds (s). If the unit is omitted then seconds are the default. *number* can be either an integer constant or a :*variablename* reference to a variable having an integer value.

Example:

```
\sleep 10 ms
```

\setshell *varname command* [*argument* ...]

Sets variable *varname* to the result of the shell command *command* with the given *argument*(s). The command must return an integer value through its standard output.

command and each *argument* can be either a text constant or a :*variablename* reference to a variable. If you want to use an *argument* starting with a colon, write an additional colon at the beginning of *argument*.

Example:

```
\setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_with_colon
```

\shell *command* [*argument* ...]

Same as \setshell, but the result of the command is discarded.

Example:

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

\startpipeline
\endpipeline

These commands delimit the start and end of a pipeline of SQL statements. In pipeline mode, statements are sent to the server without waiting for the results of previous statements. See Section 34.5 for more details. Pipeline mode requires the use of extended query protocol.

Built−in Operators

The arithmetic, bitwise, comparison and logical operators listed in Table 289 are built into pgbench and may be used in expressions appearing in \set. The operators are listed in increasing precedence order. Except as noted, operators taking two numeric inputs will produce a double value if either input is double, otherwise they produce an integer result.

Table 289. pgbench Operators

Built−In Functions

The functions listed in Table 290 are built into pgbench and may be used in expressions appearing in \set.

Table 290. pgbench Functions

The random function generates values using a uniform distribution, that is all the values are drawn within the specified range with equal probability. The random_exponential, random_gaussian and random_zipfian functions require an additional double parameter which determines the precise shape of the distribution.

- For an exponential distribution, *parameter* controls the distribution by truncating a quickly−decreasing exponential distribution at *parameter*, and then projecting onto integers between the bounds. To be precise, with

  ```
  f(x) = exp(−parameter * (x − min) / (max − min + 1)) / (1 − exp(−parameter))
  ```

  Then value *i* between *min* and *max* inclusive is drawn with probability: f(i) − f(i + 1).

  Intuitively, the larger the *parameter*, the more frequently values close to *min* are accessed, and the less frequently values close to *max* are accessed. The closer to 0 *parameter* is, the flatter (more uniform) the access distribution. A crude approximation of the distribution is that the most frequent 1% values in the range, close to *min*, are drawn *parameter*% of the time. The *parameter* value must be strictly positive.

- For a Gaussian distribution, the interval is mapped onto a standard normal distribution (the classical bell−shaped Gaussian curve) truncated at −parameter on the left and +parameter on the right. Values in the middle of the interval are more likely to be drawn. To be precise, if PHI(x) is the cumulative distribution function of the standard normal distribution, with mean mu defined as (max + min) / 2.0, with

  ```
  f(x) = PHI(2.0 * parameter * (x − mu) / (max − min + 1)) /
         (2.0 * PHI(parameter) − 1)
  ```

  then value *i* between *min* and *max* inclusive is drawn with probability: f(i + 0.5) − f(i − 0.5). Intuitively, the larger the *parameter*, the more frequently values close to the middle of the interval are drawn, and the less frequently values close to the *min* and *max* bounds. About 67% of values are drawn from the middle 1.0 / parameter, that is a relative 0.5 / parameter around the mean, and 95% in the middle 2.0 / parameter, that is a relative 1.0 / parameter around the mean; for instance, if *parameter* is 4.0, 67% of values are drawn from the middle quarter (1.0 / 4.0) of the interval (i.e., from 3.0 / 8.0 to 5.0 / 8.0) and 95% from the middle half (2.0 / 4.0) of the interval (second and third quartiles). The minimum allowed *parameter* value is 2.0.

- random_zipfian generates a bounded Zipfian distribution. *parameter* defines how skewed the distribution is. The larger the *parameter*, the more frequently values closer to the beginning of the interval are drawn. The distribution is such that, assuming the range starts from 1, the ratio of the probability of drawing *k* versus drawing *k+1* is ((*k*+1)/*k*)***parameter*. For example, random_zipfian(1, ..., 2.5) produces the value 1 about (2/1)**2.5 = 5.66 times more frequently than 2, which itself is produced (3/2)**2.5 = 2.76 times more frequently than 3, and so on.

  pgbench's implementation is based on "Non−Uniform Random Variate Generation", Luc Devroye, p. 550−551, Springer 1986. Due to limitations of that algorithm, the *parameter* value is restricted to the range [1.001, 1000].

Note

When designing a benchmark which selects rows non−uniformly, be aware that the rows chosen may be correlated with other data such as IDs from a sequence or the physical row ordering, which may skew performance measurements.

To avoid this, you may wish to use the **permute** function, or some other additional step with similar effect, to shuffle the selected rows and remove such correlations.

Hash functions hash, hash_murmur2 and hash_fnv1a accept an input value and an optional seed parameter. In case the seed isn't provided the value of :default_seed is used, which is initialized randomly unless set by the command−line −D option.

permute accepts an input value, a size, and an optional seed parameter. It generates a pseudorandom permutation of integers in the range [0, size), and returns the index of the input value in the permuted values. The permutation chosen is parameterized by the seed, which defaults to :default_seed, if not specified. Unlike the hash functions, permute ensures that there are no collisions or holes in the output values. Input values outside the interval are interpreted modulo the size. The function raises an error if the size is not positive. **permute** can be used to scatter the distribution of non−uniform random functions such as random_zipfian or random_exponential so that values drawn more often are not trivially correlated. For instance, the following pgbench script simulates a possible real world workload typical for social media and blogging platforms where a few accounts generate excessive load:

```
\set size 1000000
\set r random_zipfian(1, :size, 1.07)
\set k 1 + permute(:r, :size)
```

In some cases several distinct distributions are needed which don't correlate with each other and this is when the optional seed parameter comes in handy:

```
\set k1 1 + permute(:r, :size, :default_seed + 123)
\set k2 1 + permute(:r, :size, :default_seed + 321)
```

A similar behavior can also be approximated with **hash**:

```
\set size 1000000
\set r random_zipfian(1, 100 * :size, 1.07)
\set k 1 + abs(hash(:r)) % :size
```

However, since **hash** generates collisions, some values will not be reachable and others will be more frequent than expected from the original distribution.

As an example, the full definition of the built−in TPC−B−like transaction is:

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(−5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

This script allows each iteration of the transaction to reference different, randomly−chosen rows. (This example also shows why it's important for each client session to have its own variables — otherwise they'd not be independently touching different rows.)

Per−Transaction Logging

With the **−l** option (but without the **−−aggregate−interval** option), pgbench writes information about each transaction to a log file. The log file will be named *prefix.nnn*, where *prefix* defaults to pgbench_log, and *nnn* is the PID of the pgbench process. The prefix can be changed by using the **−−log−prefix** option. If the **−j** option is 2 or higher, so that there are multiple worker threads, each will have its own log file. The first worker will use the same name for its log file as in the standard single worker case. The additional log files for the other workers will be named *prefix.nnn.mmm*, where *mmm* is a sequential number for each worker starting with 1.

Each line in a log file describes one transaction. It contains the following space−separated fields:

client_id
: identifies the client session that ran the transaction

transaction_no
: counts how many transactions have been run by that session

time
: transaction's elapsed time, in microseconds

script_no
: identifies the script file that was used for the transaction (useful when multiple scripts are specified with **−f** or **−b**)

time_epoch
: transaction's completion time, as a Unix−epoch time stamp

time_us
: fractional−second part of transaction's completion time, in microseconds

schedule_lag
: transaction start delay, that is the difference between the transaction's scheduled start time and the time it actually started, in microseconds (present only if **−−rate** is specified)

retries
: count of retries after serialization or deadlock errors during the transaction (present only if **−−max−tries** is not equal to one)

When both **−−rate** and **−−latency−limit** are used, the *time* for a skipped transaction will be reported as skipped. If the transaction ends with a failure, its *time* will be reported as failed. If you use the **−−failures−detailed** option, the *time* of the failed transaction will be reported as serialization or deadlock depending on the type of failure (see Failures and Serialization/Deadlock Retries for more information).

Here is a snippet of a log file generated in a single−client run:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Another example with −−rate=100 and −−latency−limit=5 (note the additional *schedule_lag* column):

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

In this example, transaction 82 was late, because its latency (6.173 ms) was over the 5 ms limit. The next two transactions were skipped, because they were already late before they were even started.

The following example shows a snippet of a log file with failures and retries, with the maximum number of tries set to 10 (note the additional *retries* column):

```
3 0 47423 0 1499414498 34501 3
3 1 8333 0 1499414498 42848 0
3 2 8358 0 1499414498 51219 0
4 0 72345 0 1499414498 59433 6
```

```
1 3 41718 0 1499414498 67879 4
1 4 8416 0 1499414498 76311 0
3 3 33235 0 1499414498 84469 3
0 0 failed 0 1499414498 84905 9
2 0 failed 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

If the **−−failures−detailed** option is used, the type of failure is reported in the *time* like this:

```
3 0 47423 0 1499414498 34501 3
3 1 8333 0 1499414498 42848 0
3 2 8358 0 1499414498 51219 0
4 0 72345 0 1499414498 59433 6
1 3 41718 0 1499414498 67879 4
1 4 8416 0 1499414498 76311 0
3 3 33235 0 1499414498 84469 3
0 0 serialization 0 1499414498 84905 9
2 0 serialization 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

When running a long test on hardware that can handle a lot of transactions, the log files can become very large. The **−−sampling−rate** option can be used to log only a random sample of transactions.

Aggregated Logging

With the **−−aggregate−interval** option, a different format is used for the log files. Each log line describes one aggregation interval. It contains the following space−separated fields:

interval_start
start time of the interval, as a Unix−epoch time stamp

num_transactions
number of transactions within the interval

sum_latency
sum of transaction latencies

sum_latency_2
sum of squares of transaction latencies

min_latency
minimum transaction latency

max_latency
maximum transaction latency

sum_lag
sum of transaction start delays (zero unless **−−rate** is specified)

sum_lag_2
sum of squares of transaction start delays (zero unless **−−rate** is specified)

min_lag
minimum transaction start delay (zero unless **−−rate** is specified)

max_lag
maximum transaction start delay (zero unless **−−rate** is specified)

skipped
number of transactions skipped because they would have started too late (zero unless **−−rate** and **−−latency−limit** are specified)

retried
number of retried transactions (zero unless **−−max−tries** is not equal to one)

retries
number of retries after serialization or deadlock errors (zero unless **−−max−tries** is not equal to one)

serialization_failures
number of transactions that got a serialization error and were not retried afterwards (zero unless **−−failures−detailed** is specified)

deadlock_failures
number of transactions that got a deadlock error and were not retried afterwards (zero unless **−−failures−detailed** is specified)

Here is some example output generated with these options:

```
pgbench −−aggregate−interval=10 −−time=20 −−client=10 −−log −−rate=1000 −−latency−limit=10 −−failures−detailed

1650260552 5178 26171317 177284491527 1136 44462 2647617 7321113867 0 9866 64 7564 28340 4148 0
1650260562 4808 25573984 220121792172 1171 62083 3037380 9666800914 0 9998 598 7392 26621 4527 0
```

Notice that while the plain (unaggregated) log format shows which script was used for each transaction, the aggregated format does not. Therefore if you need per−script data, you need to aggregate the data on your own.

Per−Statement Report

With the **−r** option, pgbench collects the following statistics for each statement:

- latency — elapsed transaction time for each statement. pgbench reports an average value of all successful runs of the statement.
- The number of failures in this statement. See Failures and Serialization/Deadlock Retries for more information.
- The number of retries after a serialization or a deadlock error in this statement. See Failures and Serialization/Deadlock Retries for more information.

The report displays retry statistics only if the **−−max−tries** option is not equal to 1.

All values are computed for each statement executed by every client and are reported after the benchmark has finished.

For the default script, the output will look similar to this:

```
starting vacuum...end.
transaction type: <builtin: TPC−B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
number of transactions above the 50.0 ms latency limit: 1311/10000 (13.110 %)
latency average = 28.488 ms
latency stddev = 21.009 ms
initial connection time = 69.068 ms
tps = 346.224794 (without initial connection time)
statement latencies in milliseconds and failures:
  0.012  0  \set aid random(1, 100000 * :scale)
  0.002  0  \set bid random(1, 1 * :scale)
  0.002  0  \set tid random(1, 10 * :scale)
  0.002  0  \set delta random(−5000, 5000)
  0.319  0  BEGIN;
  0.834  0  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  0.641  0  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
 11.126  0  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
 12.961  0  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
  0.634  0  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMEST
  1.957  0  END;
```

Another example of output for the default script using serializable default transaction isolation level (**PGOPTIONS='−c default_transaction_isolation=serializable' pgbench ...**):

```
starting vacuum...end.
transaction type: <builtin: TPC−B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 10
number of transactions per client: 1000
number of transactions actually processed: 6317/10000
number of failed transactions: 3683 (36.830%)
number of transactions retried: 7667 (76.670%)
total number of retries: 45339
number of transactions above the 50.0 ms latency limit: 106/6317 (1.678 %)
latency average = 17.016 ms
latency stddev = 13.283 ms
initial connection time = 45.017 ms
tps = 186.792667 (without initial connection time)
statement latencies in milliseconds, failures and retries:
  0.006     0      0  \set aid random(1, 100000 * :scale)
  0.001     0      0  \set bid random(1, 1 * :scale)
  0.001     0      0  \set tid random(1, 10 * :scale)
  0.001     0      0  \set delta random(−5000, 5000)
  0.385     0      0  BEGIN;
  0.773     0      1  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  0.624     0      0  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  1.098   320   3762  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
  0.582  3363  41576  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
  0.465     0      0  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TI
  1.933     0      0  END;
```

If multiple script files are specified, all statistics are reported separately for each script file.

Note that collecting the additional timing information needed for per−statement latency computation adds some overhead. This will slow average execution speed and lower the computed TPS. The amount of slowdown varies significantly depending on platform and hardware. Comparing average TPS values with and without latency reporting enabled is a good way to measure if the timing overhead is significant.

Failures and Serialization/Deadlock Retries

When executing pgbench, there are three main types of errors:

- Errors of the main program. They are the most serious and always result in an immediate exit from pgbench with the corresponding error message. They include:
  - errors at the beginning of pgbench (e.g. an invalid option value);

  - errors in the initialization mode (e.g. the query to create tables for built−in scripts fails);
  - errors before starting threads (e.g. could not connect to the database server, syntax error in the meta command, thread creation failure);
  - internal pgbench errors (which are supposed to never occur...).
- Errors when the thread manages its clients (e.g. the client could not start a connection to the database server / the socket for connecting the client to the database server has become invalid). In such cases all clients of this thread stop while other threads continue to work.
- Direct client errors. They lead to immediate exit from pgbench with the corresponding error message only in the case of an internal pgbench error (which are supposed to never occur...). Otherwise in the worst case they only lead to the abortion of the failed client while other clients continue their run (but some client errors are handled without an abortion of the client and reported separately, see below). Later in this section it is assumed that the discussed errors are only the direct client errors and they are not internal pgbench errors.

A client's run is aborted in case of a serious error; for example, the connection with the database server was lost or the end of script was reached without completing the last transaction. In addition, if execution of an SQL or meta command fails for reasons other than serialization or deadlock errors, the client is aborted. Otherwise, if an SQL command fails with serialization or deadlock errors, the client is not aborted. In such cases, the current transaction is rolled back, which also includes setting the client variables as they were before the run of this transaction (it is assumed that one transaction script contains only one transaction; see What Is the "Transaction" Actually Performed in pgbench? for more information). Transactions with serialization or deadlock errors are repeated after rollbacks until they complete successfully or reach the maximum number of tries (specified by the **−−max−tries** option) / the maximum time of retries (specified by the **−−latency−limit** option) / the end of benchmark (specified by the **−−time** option). If the last trial run fails, this transaction will be reported as failed but the client is not aborted and continues to work.

Note

Without specifying the **−−max−tries** option, a transaction will never be retried after a serialization or deadlock error because its default value is 1. Use an unlimited number of tries (−−max−tries=0) and the **−−latency−limit** option to limit only the maximum time of tries. You can also use the **−−time** option to limit the benchmark duration under an unlimited number of tries.

Be careful when repeating scripts that contain multiple transactions: the script is always retried completely, so successful transactions can be performed several times.

Be careful when repeating transactions with shell commands. Unlike the results of SQL commands, the results of shell commands are not rolled back, except for the variable value of the **\setshell** command.

The latency of a successful transaction includes the entire time of transaction execution with rollbacks and retries. The latency is measured only for successful transactions and commands but not for failed transactions or commands.

The main report contains the number of failed transactions. If the **−−max−tries** option is not equal to 1, the main report also contains statistics related to retries: the total number of retried transactions and total number of retries. The per−script report inherits all these fields from the main report. The per−statement report displays retry statistics only if the **−−max−tries** option is not equal to 1.

If you want to group failures by basic types in per−transaction and aggregation logs, as well as in the main and per−script reports, use the **−−failures−detailed** option. If you also want to distinguish all errors and failures (errors without retrying) by type including which limit for retries was exceeded and how much it was exceeded by for the serialization/deadlock failures, use the **−−verbose−errors** option.

Good Practices

It is very easy to use pgbench to produce completely meaningless numbers. Here are some guidelines to help you get useful results.

In the first place, *never* believe any test that runs for only a few seconds. Use the **−t** or **−T** option to make the run last at least a few minutes, so as to average out noise. In some cases you could need hours to get numbers that are reproducible. It's a good idea to try the test run a few times, to find out if your numbers are reproducible or not.

For the default TPC−B−like test scenario, the initialization scale factor (**−s**) should be at least as large as the largest number of clients you intend to test (**−c**); else you'll mostly be measuring update contention. There are only **−s** rows in the pgbench_branches table, and every transaction wants to update one of them, so **−c** values in excess of **−s** will undoubtedly result in lots of transactions blocked waiting for other transactions.

The default test scenario is also quite sensitive to how long it's been since the tables were initialized: accumulation of dead rows and dead space in the tables changes the results. To understand the results you must keep track of the total number of updates and when vacuuming happens. If autovacuum is enabled it can result in unpredictable changes in measured performance.

A limitation of pgbench is that it can itself become the bottleneck when trying to test a large number of client sessions. This can be alleviated by running pgbench on a different machine from the database server, although low network latency will be essential. It might even be useful to run several pgbench instances concurrently, on several client machines, against the same database server.

Security

If untrusted users have access to a database that has not adopted a secure schema usage pattern, do not run pgbench in that database. pgbench uses unqualified names and does not manipulate the search path.