Performance Tuning PostgreSQL

by  Frank Wiles

Introduction

PostgreSQL is the most advanced and flexible Open Source SQL database today. With this power and flexibility comes a problem. How do the PostgreSQL developers tune the default configuration for everyone? Unfortunately the answer is they can't.

The problem is that every database is not only different in its design, but also its requirements. Some systems are used to log mountains of data that is almost never queried. Others have essentially static data that is queried constantly, sometimes feverishly. Most systems however have some, usually unequal, level of reads and writes to the database. Add this little complexity on top of your totally unique table structure, data, and hardware configuration and hopefully you begin to see why tuning can be difficult.

The default configuration PostgreSQL ships with is a very solid configuration aimed at everyone's best guess as to how an "average" database on "average" hardware should be setup. This article aims to help PostgreSQL users of all levels better understand PostgreSQL performance tuning.

Understanding the process

The first step to learning how to tune your PostgreSQL database is to understand the life cycle of a query. Here are the steps of a query:

Transmission of query string to database backend

Parsing of query string

Planning of query to optimize retrieval of data

Retrieval of data from hardware

Transmission of results to client

them into the database as a stored procedure and cut the data transfer down to a minimum.

Once the SQL query is inside the database server it is parsed into tokens. This step can also be minimized by using stored procedures.

The planning of the query is where PostgreSQL really starts to do some work. This stage checks to see if the query is already prepared if your version of PostgreSQL and client library support this feature. It also analyzes your SQL to determine what the most efficient way of retrieving your data is. Should we use an index and if so which one? Maybe a hash join on those two tables is appropriate? These are some of the decisions the database makes at this point of the process. This step can be eliminated if the query is previously prepared.

Now that PostgreSQL has a plan of what it believes to be the best way to retrieve the data, it is time to actually get it. While there are some tuning options that help here, this step is mostly effected by your hardware configuration.

And finally the last step is to transmit the results to the client. While there aren't any real tuning options for this step, you should be aware that all of the data that you are returning is pulled from the disk and sent over the wire to your client. Minimizing the number of rows and columns to only those that are necessary can often increase your performance.

General Tuning

There are several postmaster options that can be set that drastically affect performance, below is a list of the most commonly used and how they effect performance:

- *max_connections = <num>* — This option sets the maximum number of database backend to have at any one time. Use this feature to ensure that you do not launch so many backends that you begin swapping to disk and kill the performance of all the children. Depending on your application it may be better to deny the connection entirely rather than degrade the performance of all of the other children.
- *shared_buffers = <num>* — Editing this option is the simplest way to improve the performance of your database server. The default is pretty low for most modern hardware. General wisdom says that this should be set to roughly 25% of available RAM on the system. Like most of the options I will outline here you will simply need to try them at different levels (both up and down) and see how well it works on your particular system. Most people find that setting it larger than a third starts to degrade performance.
- *effective_cache_size = <num>* — This value tells PostgreSQL's optimizer how much memory PostgreSQL has available for caching data and helps in determing whether or not it use an index or not. The larger the value increases the likely hood of using an index. This should be set to the amount of memory allocated to shared_buffers plus the amount of OS cache available. Often this is more than 50% of the total system memory.

parameter, but a per operation one. So if a complex query has several sort operations in it it will use multiple work_mem units of memory. Not to mention that multiple backends could be doing this at once. This query can often lead your database server to swap if the value is too large. This option was previously called *sort_mem* in older versions of PostgreSQL.

- *max_fsm_pages = <num>* — This option helps to control the free space map. When something is deleted from a table it isn't removed from the disk immediately, it is simply marked as "free" in the free space map. The space can then be reused for any new INSERTs that you do on the table. If your setup has a high rate of DELETEs and INSERTs it may be necessary increase this value to avoid table bloat.
- *fsync = <boolean>* — This option determines if all your WAL pages are fsync()'ed to disk before a transactions is committed. Having this on is safer, but can reduce write performance. If fsync is not enabled there is the chance of unrecoverable data corruption. Turn this off at your own risk.
- *commit_delay = <num>* and *commit_siblings = <num>* — These options are used in concert to help improve performance by writing out multiple transactions that are committing at once. If there are commit_siblings number of backends active at the instant your transaction is committing then the server waiting commit_delay microseconds to try and commit multiple transactions at once.
- *random_page_cost = <num>* — random_page_cost controls the way PostgreSQL views non-sequential disk reads. A higher value makes it more likely that a sequential scan will be used over an index scan indicating that your server has very fast disks.

Note that many of these options consume shared memory and it will probably be necessary to increase the amount of shared memory allowed on your system to get the most out of these options.

If this is still confusing to you, Revolution Systems does offer a PostgreSQL Tuning Service

Hardware Issues

Obviously the type and quality of the hardware you use for your database server drastically impacts the performance of your database. Here are a few tips to use when purchasing hardware for your database server (in order of importance):

- **RAM** — The more RAM you have the more disk cache you will have. This greatly impacts performance considering memory I/O is thousands of times faster than disk I/O.
- **Disk types** — Obviously fast Ultra-320 SCSI disks are your best option, however high end SATA drives are also very good. With SATA each disk is substantially cheaper and with that you can afford more spindles than with SCSI on the same budget.
- **Disk configuration** — The optimum configuration is RAID 1+0 with as many disks as possible and with your transaction log (pg_xlog) on a separate disk (or stripe) all by itself. RAID 5 is not a very good option for databases unless you have more than 6 disks in your volume. With

drives.

- **CPUs** — The more CPUs the better, however if your database does not use many complex functions your money is best spent on more RAM or a better disk subsystem.

In general the more RAM and disk spindles you have in your system the better it will perform. This is because with the extra RAM you will access your disks less. And the extra spindles help spread the reads and writes over multiple disks to increase throughput and to reduce drive head congestion.

Another good idea is to separate your application code and your database server onto different hardware. Not only does this provide more hardware dedicated to the database server, but the operating system's disk cache will contain more PostgreSQL data and not other various application or system data this way.

For example, if you have one web server and one database server you can use a cross-over cable on a separate ethernet interface to handle just the web server to database network traffic to ensure you reduce any possible bottlenecks there. You can also obviously create an entirely different physical network for database traffic if you have multiple servers that access the same database server.

Useful Tuning Tools

The most useful tool in tuning your database is the SQL command EXPLAIN ANALYZE. This allows you to profile each SQL query your application performs and see exactly how the PostgreSQL planner will process the query. Let's look at a short example, below is a simple table structure and query.

```
CREATE TABLE authors (
    id    int4 PRIMARY KEY,
    name  varchar
);

CREATE TABLE books (
    id          int4 PRIMARY KEY,
    author_id   int4,
    title       varchar
);
```

If we use the query:

```
FROM books, authors
WHERE books.author_id=16 and authors.id = books.author_id
ORDER BY books.title;
```

You will get output similar to the following:

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=29.71..29.73 rows=6 width=64) (actual time=0.189..16.233 rows=7 loops=1)
   Sort Key: books.title
   ->  Nested Loop  (cost=0.00..29.63 rows=6 width=64) (actual time=0.068..0.129 rows=7 loops=1)
         ->  Index Scan using authors_pkey on authors  (cost=0.00..5.82 rows=1 width=36) (actual time=0.029..0.033 rows=1 loops=1)
               Index Cond: (id = 16)
         ->  Seq Scan on books  (cost=0.00..23.75 rows=6 width=36) (actual time=0.026..0.052 rows=7 loops=1)
               Filter: (author_id = 16)
 Total runtime: 16.386 ms
```

You need to read this output from bottom to top when analyzing it. The first thing PostgreSQL does is do a sequence scan on the books table looking at each author_id column for values that equal 16. Then it does an index scan of the authors table, because of the implicit index created by the PRIMARY KEY options. Then finally the results are sorted by books.title.

The values you see in parenthesis are the estimated and actual cost of that portion of the query. The closer together the estimate and the actual costs are the better performance you will typically see.

Now, let's change the structure a little bit by adding an index on books.author_id to avoid the sequence scan with this command:

```
CREATE INDEX books_idx1 on books(author_id);
```

If you rerun the query again, you won't see any noticeable change in the output. This is because PostgreSQL has not yet re-analyzed the data and determined that the new index may help for this query. This can be solved by running:

```
ANALYZE books;
```

However, in this small test case I'm working with the planner still favors the sequence scan because there aren't very many rows in my books table. If a query is going to return a large portion of a table then the planner chooses a sequence scan over an index because it is actually faster. You can also force PostgreSQL to favor index scans over sequential scans by setting the configuration parameter *enable_seqscan* to off. This doesn't remove all sequence scans, since some tables may not have an index, but it does force the planner's hand into always using an index scan when it is available. This is probably best done by sending the command *SET*

intended for every day use.

Typically the best way to optimize your queries is to use indexes on specific columns and combinations of columns to correspond to often used queries. Unfortunately this is done by trial and error. You should also note that increasing the number of indexes on a table increases the number of write operations that need to be performed for each INSERT and UPDATE. So don't do anything silly and just add indexes for each column in each table.

You can help PostgreSQL do what you want by playing with the level of statistics that are gathered on a table or column with the command:

```
ER TABLE <table> ALTER COLUMN <column> SET STATISTICS <number>;
```

This value can be a number between 0 and 1000 and helps PostgreSQL determine what level of statistics gathering should be performed on that column. This helps you to control the generated query plans without having slow vacuum and analyze operations because of generating large amounts of stats for all tables and columns.

Another useful tool to help determine how to tune your database is to turn on query logging. You can tell PostgreSQL which queries you are interested in logging via the *log_statement* configuration option. This is very useful in situations where you many users executing ad hoc queries to your system via something like Crystal Reports or via psql directly.

Database Design and Layout

Sometimes the design and layout of your database affects performance. For example, if you have an employee database that looks like this:

```
CREATE TABLE employees (
    id              int4 PRIMARY KEY,
    active          boolean,
    first_name      varchar,
    middle_name     varchar,
    last_name       varchar,
    ssn             varchar,
    address1        varchar,
    address2        varchar,
    city            varchar,
    state           varchar(2),
    zip             varchar,
    home_phone      varchar,
    work_phone      varchar,
```

```
    business_email    varchar,
    personal_email    varchar,
    salary            int4,
    vacation_days     int2,
    sick_days         int2,
    employee_number   int4,
    office_addr_1     varchar,
    office_addr_2     varchar,
    office_city       varchar,
    office_state      varchar(2),
    office_zip        varchar,
    department        varchar,
    title             varchar,
    supervisor_id     int4
);
```

This design is easy to understand, but isn't very good on several levels. While it will depend on your particular application, in most cases you won't need to access all of this data at one time. In portions of your application that deal with HR functions you are probably only interested in their name, salary, vacation time, and sick days. However, if the application displays an organization chart it would only be concerned with the department and supervisor_id portions of the table.

By breaking up this table into smaller tables you can get more efficient queries since PostgreSQL has less to read through, not to mention better functionality. Below is one way to make this structure better:

```
CREATE TABLE employees (
    id              int4 PRIMARY KEY,
    active          boolean,
    employee_number int4,
    first_name      varchar,
    middle_name     varchar,
    last_name       varchar,
    department      varchar,
    title           varchar,
    email           varchar
);

CREATE TABLE employee_address (
```

```
    address_1       varchar,
    address_2       varchar,
    city            varchar,
    state           varchar(2),
    zip             varchar
);

CREATE TABLE employee_number_type (
    id              int4 PRIMARY KEY,
    type            varchar
);

CREATE TABLE employee_number (
    id              int4 PRIMARY KEY,
    employee_id     int4,
    type_id         int4,
    number          varchar
);

CREATE TABLE employee_hr_info (
    id              int4 PRIMARY KEY,
    employee_id     int4,
    ssn             varchar,
    salary          int4,
    vacation_days   int2,
    sick_days       int2
);
```

With this table structure the data associated with an employee is broken out into logical groupings. The main table contains the most frequently used information and the other tables store all of the rest of the information. The added benefit of this layout is that you can have any number of phone numbers and addresses associated with a particular employee now.

Another useful tip is to use partial indexes on columns where you typically query a certain value more often than another. Take for example the employee table above. You're probably only displaying active employees throughout the majority of the application, but creating a partial index on that column where the value is true can help speed up the query and may help the planner to choose to use the index in cases where it otherwise would not. You can create a partial index like this:

row is associated with an employee, maybe in some trouble ticket like system. In that type of application you would probably have a 'View Unassigned Tickets' portion of the application which would benefit from a partial index such as this:

```
ATE INDEX tickets_idx1 ON tickets(employee_id) WHERE employee_id IS NULL;
```

Application Development

There are many different ways to build applications which use a SQL database, but there are two very common themes that I will call *stateless* and *stateful*. In the area of performance there are different issues that impact each.

Stateless is typically the access type used by web based applications. Your software connects to the database, issues a couple of queries, returns to results to the user, and disconnects. The next action the users takes restarts this process with a new connect, new set of queries, etc.

Stateful applications are typically non-web based user interfaces where an application initiates a database connection and holds it open for the duration the application is in use.

Stateless Applications

In web based applications each time something is requested by the user , the application initiates a new database connection. While PostgreSQL has a very short connection creation time and in general it is not a very expensive operation, it is best to use some sort of database connection pooling method to get maximum performance.

There are several ways to accomplish database connection pooling, here is a short list of common ones:

- Pgpool is a small server that you run on the same server as your clients that will pool database connections to some local or remote server. The application simply points at the pgpool instance instead of the normal postmaster. From the application's perspective nothing has changed as the connection pooling is hidden from it.
- In a mod_perl environment you can use Apache::DBI to handle database connection pooling inside of Apache itself.
- SQLRelay is another db connection manager that is somewhat database agnostic. It works with with several databases other than PostgreSQL.
- You can always write a small bit of code to do this for you yourself, but I would highly recommend using an already developed solution to reduce the amount of debugging you have to do.

It should be noted that in a few bizarre instances I've actually seen database connection pooling reduce the performance of web based applications. At a certain point the cost of handling the

When building stateful applications you should look into using database cursors via the DECLARE command. A cursor allows you to plan and execute a query, but only pull back the data as you need it, for example one row at a time. This can greatly increase the snappiness of the UI.

General Application Issues

These issues typically effect both stateful and stateless applications in the same fashion. One good technique is to use server side prepared queries for any queries you execute often. This reduces the overall query time by caching the query plan for later use.

It should be noted however if you prepare a query in advance using placeholder values (such as 'column_name = ?') then the planner will not always be able to choose the best plan. For example, your query has a placeholder for the boolean column 'active' and you have a partial index on false values the planner won't use it because it cannot be sure the value passed in on execution will be true or false.

You can also obviously utilize stored procedures here to reduce the transmit, parse, and plan portions of the typical query life cycle. It is best to profile your application and find commonly used queries and data manipulations and put them into a stored procedure.

Other Useful Resources

Here is a short list of other items that may be of help.

- PostgreSQL Homepage — The obvious place for all things PostgreSQL.
- psql-performance mailing list — This PostgreSQL mailing list is focused on performance related questions and discussions.
- PostgreSQL Tuning Service
- PostgreSQL Support Service

Recommended Books

- PostgreSQL 9.0 High Performance
- Beginning Databases with PostgreSQL: From Novice to Professional
- PostgreSQL 9 Admin Cookbook
- Practical PostgreSQL (O'Reilly Unix)
- PostgreSQL: Introduction and Concepts

Frank Wiles switched to using PostgreSQL as his primary database system over 15 years ago and has never looked back. He has used PostgreSQL in a variety of situations, most often however coupled with Python and Django to build high performance browser based applications. He has published several articles and a book on topics ranging from systems administration to application development. He can be reached at frank@revsys.com.

Let's work together.

sales@revsys.com Contact us