Technologia i rozwiązania

Wysoko wydajny PostgreSQL 9.0

Poznaj najlepsze techniki zwiększania wydajności PostgreSQL i sprawdzone rozwiązania najczęściej spotykanych problemów!

- Jak dobierać komponenty serwera, aby maksymalnie wykorzystać jego możliwości?
- Jak przeprowadzać testy wydajności całego systemu, od sprzętu po aplikację?
- Jak skutecznie indeksować bazę danych i optymalizować zapytania?

Helion

Gregory Smith

PACKT PUBLISHING

Wysoko wydajny PostgreSQL 9.0

Autor: Gregory Smith
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-3062-2
Tytuł oryginału: PostgreSQL 9.0 High Performance
Format: 170×230, stron: 464

Poznaj najlepsze techniki zwiększania wydajności PostgreSQL i sprawdzone rozwiązania najczęściej spotykanych problemów!

- Jak dobierać komponenty serwera, aby maksymalnie wykorzystać jego możliwości?
- Jak przeprowadzać testy wydajności całego systemu, od sprzętu po aplikację?
- Jak skutecznie indeksować bazę danych i optymalizować zapytania?

Mający za sobą już ponad piętnaście lat rozwoju PostgreSQL jest dziś potężnym systemem baz danych typu open source, o sprawdzonej architekturze i reputacji narzędzia niezawodnego oraz nieprzeciętnie wydajnego. Współdziała on ze wszystkimi popularnymi systemami operacyjnymi i jest w pełni zgodny z warunkami ACID. Te zalety sprawiają, że można go używać jako magazynu danych dla aplikacji oraz jako bazy danych dla aplikacji sieciowych. Jednak osiągnięcie maksymalnej wydajności PostgreSQL nie jest wcale zadaniem łatwym, a w trakcie korzystania z jego serwerów można napotkać powtarzające się trudności, zwłaszcza gdy wzrasta obciążenie serwera, a wymagania stają się coraz większe. Jeśli zatem nie chcesz tygodniami dochodzić do właściwych rozwiązań swoich problemów – oto książka, w której znajdziesz całą potrzebną Ci wiedzę.

Masz w rękach kompletny podręcznik, przeznaczony dla średnio i bardzo zaawansowanych administratorów baz danych, którzy już używają PostgreSQL lub dopiero zamierzają to zrobić. Najpierw zapoznasz się z najnowszymi wersjami tej platformy oraz dowiesz się, jak dobierać komponenty serwera, aby optymalnie wykorzystać możliwości systemu.
Dzięki tej książce:

- Poznasz najlepsze praktyki pozwalające na obsłużenie wymagających aplikacji
- Odkryjesz, dlaczego sprzęt komputerowy nadaje się (lub nie) dla wysoko wydajnych aplikacji bazodanowych
- Zrozumiesz, na czym polegają kompromisy związane z szybkością i niezawodnością działania
- Zoptymalizujesz system operacyjny, aby osiągnąć najlepszą wydajność bazy danych
- Przeprowadzisz testy wydajności całego systemu, od sprzętu komputerowego po aplikację
- Przeanalizujesz rzeczywiste przykłady, co pozwoli Ci poznać wpływ różnych ustawień parametrów serwera na wydajność
- Będziesz skutecznie monitorować zdarzenia zachodzące na serwerze, zarówno w bazie danych, jak i poza nią
- Znajdziesz najlepsze dodatki, rozszerzające podstawowe możliwości bazy danych PostgreSQL
- Dowiesz się, jak przygotować replikację systemów za pomocą najnowszych funkcji wprowadzonych w PostgreSQL 9.0

Zoptymalizuj swój serwer PostgreSQL i unikaj problemów, które mogą zmniejszyć jego wydajność!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 6

Optymalizacja konfiguracji serwera

Podstawowe ustawienia optymalizacyjne PostgreSQL znajdują się w zwykłym pliku tekstowym o nazwie *postgresql.conf* umieszczonym w strukturze katalogów bazy danych. Struktura katalogów w systemach z rodziny Unix często określana jest przy użyciu zmiennej środowiskowej `$PGDATA`, a więc ścieżka dostępu do pliku konfiguracyjnego na tych platformach ma postać `$PGDATA/postgresql.conf`.

W rozdziale przedstawiono omówienie parametrów konfiguracyjnych; praktycznie jest to powielony ogólny format stosowany w dokumentacji dostępnej na stronie *http://www.postgresql.org/docs/current/static/runtime-config.html*. Jednak tutaj, zamiast opisywania znaczenia każdego parametru, nacisk został położony na udzielenie wskazówek dotyczących ustawiania najważniejszych wartości z perspektywy użytkownika zainteresowanego optymalizacją wydajności. Rozdział należy więc potraktować jako uzupełnienie materiału umieszczonego w oficjalnej dokumentacji, a nie jej zamiennik.

Inne źródło informacji na omawiany tutaj temat znajduje się w artykule „Tuning Your PostgreSQL Server" dostępnym na stronie *http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server*, w którym pewne informacje są identyczne z przedstawionymi w rozdziale. Struktura artykułu wiki powoduje, że jest on aktualizowany na bieżąco i może zawierać pewne informacje dotyczące przyszłych wersji bazy danych PostgreSQL, który były niedostępne w trakcie przygotowywania tego rozdziału.

Interakcja z używaną konfiguracją

Istnieje wiele sposobów modyfikacji parametrów bazy danych, nie są to jedynie edycja pliku konfiguracyjnego i ponowne uruchomienie serwera. Poznanie i zrozumienie tych sposobów ma znaczenie krytyczne zarówno dla skrócenia czasu niedostępności serwera jedynie z powodu rutynowych zmian konfiguracyjnych, jak i zagwarantowania, że czytelnik przeprowadza modyfikację właściwych parametrów, gdy zmiana ma zostać wprowadzona natychmiast.

Ustawienia domyślne i sposoby ich zerowania

Baza danych ma dwa rodzaje ustawień, które można określić jako „ustawienia domyślne" w zależności od kontekstu. Jednym z nich są ustawienia domyślne powodujące ustalenie przez serwer pewnych wartości, jeżeli użytkownik nie zmieni żadnych opcji — są to ustawienia, z którymi serwer jest uruchamiany jeszcze przed odczytaniem pliku konfiguracyjnego *postgresql.conf*. Od wydania PostgreSQL 8.4 wartości te można sprawdzić w kolumnie `boot_val` w widoku `pg_settings`. Więcej informacji na ten temat znajduje się na stronie *http://www.postgresql.org/docs/current/static/view-pg-settings.html*.

Po uruchomieniu serwera i wprowadzeniu zmian w parametrach, które również posiadają swoje wartości domyślne, powrót do wspomnianych wartości domyślnych następuje w wyniku wydania polecenia `RESET`. Działanie tego polecenia zostało omówione na stronie *http://www.postgresql.org/docs/current/static/sql-reset.html*. Te wartości domyślne można przejrzeć w kolumnie `reset_val` znajdującej się w widoku `pg_settings`.

Dozwolony kontekst do przeprowadzania zmian

Każde ustawienie konfiguracyjne ma powiązany z nim kontekst, w ramach którego może zostać zmienione. Najlepszym sposobem określenia dozwolonego kontekstu do przeprowadzania zmian jest bezpośrednie zapytanie bazy danych. W przedstawionym poniżej przykładzie pokazano po jednym wpisie dla każdego rodzaju kontekstu (w rzeczywistości po wydaniu poniższego polecenia dane wyjściowe będą zawierały wszystkie parametry serwera):

```
postgres=# select name,context from pg_settings;
             name             |  context
------------------------------+------------
archive_command               | sighup
archive_mode                  | postmaster
block_size                    | internal
log_connections               | backend
log_min_duration_statement    | superuser
search_path                   | user
```

W oficjalnym podręczniku użytkownika pole kontekstu (`context`) nie jest zbyt dobrze udokumentowane. Poniżej przedstawiono więc znaczenie różnych stosowanych ustawień w kolejności od najtrudniejszego do najłatwiejszego do zmiany.

- `internal`. Ustawienia te w dużej mierze są wewnętrznymi ustawieniami bazy danych określonymi w trakcie jej kompilacji. Wyświetlane są w celu dostarczenia użytkownikowi informacji, ale nie mogą zostać zmienione bez ponownej kompilacji serwera.
- `postmaster`. Ustawienia są uaktualniane jedynie podczas pełnego uruchamiania serwera. Do tej kategorii zaliczają się wszystkie opcje dotyczące pamięci współdzielonej.
- `sighup`. Wysłanie sygnału HUP do serwera spowoduje ponowne odczytanie pliku konfiguracyjnego *postgresql.conf*, a wszelkie zmiany wprowadzone do tego parametru będą natychmiast zastosowane. Więcej informacji na ten temat znajduje się w kolejnym punkcie zatytułowanym „Ponowne wczytywanie pliku konfiguracyjnego".
- `backend`. Ustawienia oznaczone za pomocą tej właściwości są podobne do ustawień `sighup`, z wyjątkiem faktu, że wprowadzone zmiany nie mają wpływu na już istniejące sesje bazy danych. Tylko sesje uruchomione po wprowadzeniu zmian będą stosowały nowe ustawienia. Istnieje niewiele parametrów oznaczonych omawianą właściwością, większość z nich wpływa jedynie na działania podejmowane w trakcie uruchamiania i zamykania sesji. Ostatnia opcja tej grupy (`log_connections`) nie może działać wstecz, to znaczy nie ma możliwości rozpoczęcia rejestrowania zdarzeń już nawiązanego połączenia. Zdarzenia będą rejestrowane tylko dla nowych połączeń ustanowionych po włączeniu opcji `log_connections`.
- `superuser`. Ustawienia tej grupy mogą być modyfikowane w dowolnym momencie przez każdego superużytkownika bazy danych (z reguły użytkownika, który utworzył bazę danych, czyli bardzo często „postgres"). Aktywowanie zmiany nie wymaga pełnego, ponownego wczytania pliku konfiguracyjnego. Większość ustawień tej grupy jest powiązana z rejestrowaniem w plikach dzienników zdarzeń różnych aspektów poleceń wykonywanych przez serwer.
- `user`. Te parametry poszczególni użytkownicy mogą dostosować w dowolnym momencie swojej sesji. Wprowadzone zmiany dotyczą jedynie danej sesji. Większość parametrów zmienia sposób wykonywania zapytań, co pozwala na przeprowadzenie optymalizacji w ramach sesji.

Na podstawie powyższej sesji można się przekonać, że udzielenie odpowiedzi na — wydawałoby się — proste pytanie, jaka jest bieżąca wartość `work_mem`, może być bardzo trudne, w zależności od wybranego kontekstu. Początkowo wartość ta może być wartością określoną w pliku konfiguracyjnym *postgresql.conf*, następnie może być zmieniona w wyniku ponownego wczytania konfiguracji, a na końcu znów zmieniona przez użytkownika przed wykonaniem zapytania.

Ponowne wczytywanie pliku konfiguracyjnego

Istnieją trzy sposoby pozwalające bazie danych na ponowne wczytanie konfiguracji w celu uaktualnienia wartości zaliczających się do grupy `sighup`. Jeżeli czytelnik jest połączony z bazą danych jako superużytkownik, można użyć funkcji `pg_reload_conf()` w następujący sposób:

```
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
```

Sygnał `SIGHUP` można wysłać również ręcznie za pomocą polecenia `kill` systemu Unix:

```
$ ps -eaf | grep "postgres -D"
postgres 11185     1 0 22:21 pts/0     00:00:00 /home/postgres/inst/bin/
↪postgres -D /home/postgres/data/
$ kill -HUP 11185
```

Wreszcie, sygnał `SIGHUP` można wysyłać do serwera przy użyciu narzędzia `pg_ctl`:

```
$ pg_ctl reload
server signaled
```

Niezależnie od użytego sposobu, po wysłaniu sygnału `SIGHUP` w dziennikach zdarzeń bazy danych znajdzie się następujący komunikat:

```
LOG: received SIGHUP, reloading configuration files
```

Następnie, za pomocą poleceń, takich jak `SHOW`, lub po przejrzeniu widoku `pg_settings` można potwierdzić, że zmiany zostały wprowadzone zgodnie z oczekiwaniami.

Umieszczanie ustawień w komentarzach

Co się stanie w sytuacji, gdy pewien parametr został ręcznie ustawiony, ale musi być wyłączony w działającym serwerze? Konkretna odpowiedź zależy od wersji używanego serwera. Przyjmijmy założenie, że plik *postgresql.conf* został uruchomiony wraz z ustawionym następującym parametrem:

```
checkpoint_segments = 30
```

W celu wyłączenia tego parametru trzeba wyedytować plik konfiguracyjny, a parametr umieścić w komentarzu:

```
#checkpoint_segments = 30
```

Następnie konieczne jest nakazanie serwerowi, aby ponownie wczytał plik konfiguracyjny:

```
$ pg_ctl reload
```

Parametr checkpoint_segments jest ustawieniem kontekstu sighup. Od PostgreSQL 8.3 powyższa zmiana spowoduje powrót do wartości domyślnej serwera (boot_val). W wersji 9.0 w dzienniku zdarzeń zostanie ponadto umieszczony następujący komunikat:

```
LOG: received SIGHUP, reloading configuration files
LOG: parameter "checkpoint_segments" removed from configuration file, reset
↪to default
```

Użycie wartości domyślnej można potwierdzić za pomocą polecenia SHOW:

```
$ psql -x -c "show checkpoint_segments"
-[ RECORD 1 ]-------+--
checkpoint_segments | 3
```

Jeżeli używany serwer to PostgreSQL w wersji 8.2 lub wcześniejszej, wprowadzona zmiana nie spowoduje żadnego efektu, to znaczy parametr checkpoint_segments nadal będzie miał wartość 30. Dopiero po pełnym, ponownym uruchomieniu serwera nastąpi przywrócenie wartości domyślnej, czyli 3.

Ponieważ takie zachowanie jest skomplikowane i uzależnione od używanej wersji serwera, doświadczeni administratorzy PostgreSQL zwykle dwukrotnie sprawdzają parametry, które mają zamiar zmodyfikować. Do sprawdzenia używają polecenia SHOW lub widoku pg_settings i upewniają się w ten sposób, że parametr ma oczekiwaną wartość.

W tym miejscu inną, skomplikowaną kwestią jest możliwość dołączenia dodatkowych plików konfiguracyjnych z poziomu głównego pliku *postgresql.conf*. W przypadku takiego rozwiązania skutek jest taki sam jak wstawienie zawartości określonego pliku we wskazanym miejscu. W widoku pg_settings można poznać nazwę pliku, z którego pochodzi dany parametr, wraz z numerem wiersza zawierającego aktywną wersję. Warto też pamiętać, że w przypadku wielokrotnego ustawiania tego samego parametru pod uwagę zawsze brane jest tylko ostatnie jego wystąpienie.

Ustawienia na poziomie serwera

Wprawdzie w pewnych przypadkach parametry mogą być modyfikowane w innych kontekstach, przedstawione w tym podrozdziale są tymi, które można zmodyfikować w pliku *postgresql.conf* odczytywanym przed uruchomieniem serwera.

Połączenia z bazą danych

Istnieje wiele parametrów konfiguracyjnych określających sposoby zdalnego i lokalnego nawiązywania połączenia z bazą danych. Pełna lista wspomnianych parametrów została przedstawiona na stronie *http://www.postgresql.org/docs/current/static/runtime-config-connection.html*.

listen_addresses

W każdej instalacji wymagającej obsługi połączeń pochodzących z systemów zdalnych konieczna jest modyfikacja parametru `listen_addresses` pozwalającego na obsługę takich połączeń. Domyślnie dozwolone są jedynie połączenia lokalne pochodzące od użytkownika zalogowanego do tego samego systemu, w którym znajduje się baza danych. Powszechnie stosowanym podejściem jest akceptacja połączeń przychodzących z każdego miejsca. W tym celu w głównym pliku konfiguracyjnym trzeba umieścić poniższy wiersz:

```
listen_addresses = '*'
```

Aby kontrolować, kto może nawiązać połączenie, trzeba przeprowadzić konfigurację pliku *pg_hba.conf*, co zostało omówione na stronie *http://www.postgresql.org/docs/current/static/auth-pg-hba-conf.html*. Stosowanie takiego podejścia wiąże się z pewnym problemem dotyczącym wydajności. Filtrowanie połączeń za pomocą dokładniej skonfigurowanego parametru `listen_addresses` może być znacznie bardziej efektywne niż pozwolenie na nawiązanie połączenia wszystkim klientom. Zezwolenie klientowi na połączenie, a następnie jego odrzucenie w wyniku wpisu znajdującego się w pliku konfiguracyjnym *pg_hba.conf* powoduje niepotrzebne zużycie pewnych zasobów serwera i naraża system na niebezpieczeństwo przeprowadzenia w ten sposób ataku typu odmowa usługi — DoS (ang. *Denial Of Service*) — przez złośliwego użytkownika.

W praktyce tylko niewielka liczba serwerów PostgreSQL pozwala na bezpośrednie przyjmowanie zapytań pochodzących z internetu. Normalnie są one filtrowane przez port domyślny PostgreSQL (5432) na poziomie zapory sieciowej, co stanowi najefektywniejsze podejście i jest często stosowaną implementacją mechanizmu współdzielonej ochrony także innych aplikacji. W systemie podłączonym do internetu, takim jak na przykład klastry zawierające bazy danych dla usług „w chmurach", należy upewnić się o stosowaniu wszystkich trzech warstw ochrony. Na poziomie zapory sieciowej warto określić, kto może nawiązać połączenie z serwerem, dodatkowo zmniejszyć listę nasłuchiwanych adresów za pomocą opcji `listen_addresses`, a także ograniczyć użytkownikom dostęp do bazy danych za pomocą pliku konfiguracyjnego *pg_hba.conf*.

max_connections

Jednym z parametrów, dla którego czytelnik najczęściej będzie ustawiał wartość (z reguły 100) w pliku konfiguracyjnym *postgresql.conf* wygenerowanym przez `initdb`, jest `max_connections`. Ponieważ, jak wspomniano w poprzednim rozdziale, każde połączenie wykorzystuje niewielką ilość pamięci współdzielonej, w systemach domyślnie używających niewielkiej ilości pamięci współdzielonej nawiązanie większej ilości połączeń nie zostanie nawet dozwolone. W związku z powyższym, podobnie jak w przypadku parametru `shared_buffers`, po utworzeniu klastra bazy danych i określeniu w domyślnym pliku konfiguracyjnym największej dopuszczalnej wartości (do 100) przeprowadzane są pewne badania. W praktyce ilość pamięci niewspółdzielonej używanej przez każdego klienta podczas przeprowadzania operacji, takich jak sortowanie, jest wprawdzie znacznie większa, ale ilości wykorzystywanej pamięci współdzielonej nie można pominąć.

Ważne jest, aby w tym parametrze nie ustawiać wartości wyższej niż konieczna. Istnieje wiele wad wynikających z ustawienia wyższej wartości parametru `max_connections`. Jedna z nich to marnowanie pamięci współdzielonej, ale to problem, którym należy się najmniej przejmować, ponieważ ilość pamięci współdzielonej wykorzystywanej przez każde połączenie pozostaje niewielka.

Jednak są inne zasoby wykorzystywane przez klienta, na przykład alokacja pamięci dla operacji sortowania (kontrolowana za pomocą parametru `work_mem` omówionego dalej, w tym rozdziale), która z reguły obejmuje ogromny blok pamięci. Jeżeli dozwolona będzie obsługa większej liczby połączeń, to by bezpiecznie oszczędzać pamięć, trzeba również zmniejszyć wartości wspomnianych ustawień, tak aby zminimalizować niebezpieczeństwo alokacji większej ilości pamięci niż dostępna.

> Ze względu na problemy związane z alokacją zasobów, serwery PostgreSQL w Windows mogą mieć bardzo ograniczoną liczbę obsługiwanych połączeń. Bardzo często zdarza się, że zanim wyczerpana zostanie pamięć w obszarze Desktop Heap, obsłużonych będzie jedynie około 125 połączeń. Więcej informacji na temat tego problemu oraz potencjalne sposoby jego rozwiązania można znaleźć na stronie *http://wiki.postgresql.org/wiki/Running_&_Installing_PostgreSQL_On_Native_Windows*.

Wreszcie, nawiązywanie połączeń w bazie danych PostgreSQL należy uważać za operację intensywnie wykorzystującą zasoby. Celem bazy danych nie jest działalność w charakterze komponentu nawiązującego połączenie z bazą danych, przeprowadzającego uwierzytelnianie i dotarcie do punktu, w którym zapytanie będzie wykonane jako stosunkowo niewielka operacja. Ogólnie rzecz ujmując, obciążenie związane z nawiązaniem kilkuset połączeń staje się wąskim gardłem podczas działania serwera. Dokładna liczba połączeń, po przekroczeniu której stają się one obciążeniem, zależy od używanego sprzętu i konfiguracji serwera. Jeżeli czytelnik ma zamiar obsługiwać jednocześnie tysiące zapytań, nie może zastosować podejścia, w którym każdy klient bezpośrednio nawiązuje połączenie z bazą danych. W takim przypadku najczęściej stosowanym rozwiązaniem problemu skalowalności jest użycie oprogramowania obsługującego pulę połączeń między aplikacją i bazą danych. Temat ten został omówiony w rozdziale 13.

Pamięć współdzielona

Właściwe ustawienie wartości parametrów związanych z pamięcią współdzieloną jest ważne, ponieważ ich zmiana zawsze wymaga pełnego, ponownego uruchomienia serwera bazy danych — serwer nie ma możliwości dynamicznej ponownej alokacji pamięci współdzielonej.

shared_buffers

Parametr `shared_buffers` był szczegółowo omawiany w poprzednim rozdziale.

Ustawienia Free Space Map (FSM)

Przestrzeń pozostała w wyniku operacji usuwania bądź modyfikacji danych jest umieszczana przez polecenie `VACUUM` w przestrzeni FSM (ang. *Free Space Map*). Nowe operacje alokacji wykorzystują przestrzeń pochodzącą z FSM, zamiast alokować nową przestrzeń na dysku.

Od wydania PostgreSQL 8.4 przestrzeń FSM jest przechowywana na dysku, a tym samym automatycznie skaluje swoją wielkość. W PostgreSQL do wersji 8.3 przestrzeń FSM była przechowywana w pamięci współdzielonej, co wymagało dokładnego monitorowania ilości tej pamięci i potencjalnie prowadziło do zwiększenia jej zużycia. Upewnienie się, że wartości parametrów `max_fsm_pages` i `max_fsm_relations` w pliku konfiguracyjnym są wystarczające, powinno być częścią regularnych operacji konserwacyjnych w tych wersjach serwera PostgreSQL. Operację tę można przeprowadzić ręcznie bądź wydać polecenie `VACUUM VERBOSE` przeprowadzające pomiar bieżącego użycia pamięci współdzielonej w bardziej zautomatyzowany sposób. Więcej informacji na ten temat przedstawiono w rozdziałach 5. i 7.

Rejestrowanie zdarzeń

Ogólne ustawienia dotyczące rejestracji zdarzeń są ważne, ale w pewnym sensie pozostają poza zakresem tematycznym tej książki. Czytelnik może być zmuszony do ustawienia parametrów, takich jak `log_destination`, `log_directory` i `log_filename`, w sposób zgodny z systemem oraz wymaganiami administratora używanego środowiska. Parametry te mają domyślnie ustawione rozsądne wartości pozwalające na rozpoczęcie pracy w większości systemów. W rozdziale 7. zostanie poruszony problem dostosowania omawianych parametrów w celu przeprowadzenia rejestracji zdarzeń w plikach CSV, co może być użyteczne podczas pomiaru czasu wykonywania zapytań.

W większości systemów Unix bardzo często zdarza się, że opcje dotyczące rejestracji zdarzeń bazy danych są ustawiane w skryptach uruchamiających i zatrzymujących serwer, a nie bezpośrednio w pliku konfiguracyjnym *postgresql.conf*. Jeżeli w celu ręcznego uruchomienia serwera stosowane jest polecenie `pg_ctl`, czytelnik może odkryć, że rejestracja zdarzeń następuje poprzez ich bezpośrednie wyświetlenie na ekranie. W takim przypadku, aby dowiedzieć się, o co chodzi, trzeba spojrzeć do skryptu uruchamiającego serwer w zwykły sposób (najczęściej */etc/init.d/postgresql*). W większości przypadków do polecenia `pg_ctl` wystarczy dodać opcję `-l nazwa_pliku`, która powoduje przekierowanie danych wyjściowych do standardowego położenia.

log_line_prefix

Wartość domyślna opcji `log_line_prefix` jest pusta, a to zdecydowanie niepożądane. Wartość dobra na początek będzie miała następującą postać:

```
log_line_prefix='%t:%r:%u@%d:[%p]: '
```

Powyższy parametr powoduje umieszczenie w każdym wierszu dziennika zdarzeń takich informacji jak:

- `%t`: znacznik czasu;
- `%u`: nazwa użytkownika bazy danych;
- `%r`: nazwa zdalnego komputera, z którego nawiązano połączenie;
- `%d`: nazwa bazy danych, z którą nawiązano połączenie;
- `%p`: identyfikator procesu połączenia.

Na początku może nie być takie oczywiste, po co zostały zastosowane powyższe wartości domyślne, a zwłaszcza identyfikator procesu. Jednak po próbie rozwiązania kilku problemów związanych z wydajnością konieczność zapisania w pliku dziennika zdarzeń wymienionych informacji stanie się bardziej oczywista i czytelnik będzie zadowolony z faktu posiadania tych informacji.

Inne podejście warte rozważenia to ustawienie wartości parametru `log_line_prefix` w taki sposób, aby dzienniki zdarzeń były zgodne z programem pgFouine, omówionym w rozdziale 7. Jest to rozsądny, przeznaczony do ogólnego rejestrowania zdarzeń prefiks, a wiele witryn i tak ostatecznie stosuje pewne mechanizmy analizy zapytań.

log_statement

Oto opcje do zastosowania w tym parametrze:

- `none`. Nie są rejestrowane żadne informacje na poziomie poleceń.
- `ddl`. Rejestrowane są jedynie polecenia **DDL** (ang. *Data Definition Language*), na przykład `CREATE` lub `DROP`. Takie ustawienie można zastosować nawet w systemie produkcyjnym i jest przydatne podczas wychwytywania najważniejszych zmian przypadkowo lub celowo wprowadzonych przez administratorów.
- `mod`. Rejestrowane są wszystkie polecenia modyfikujące wartość, czyli praktycznie wszystkie, poza prostymi poleceniami `SELECT`. Jeżeli obciążenie w serwerze to przede wszystkim polecenia `SELECT` przeprowadzające niewielką ilość zmian w danych, praktycznym rozwiązaniem może być pozostawienie tej opcji włączonej przez cały czas.
- `all`. Rejestrowane są wszystkie polecenia. Ogólnie rzecz ujmując, pozostawienie tej opcji włączonej w serwerze produkcyjnym jest niepraktyczne, ze względu na obciążenie powodowane przez operację rejestrowania zdarzeń. Gdy jednak serwer jest na tyle potężny, aby poradzić sobie z takim obciążeniem, pozostawienie tej opcji włączonej przez cały czas może okazać się pomocne.

Rejestrowanie poleceń to użyteczna technika pozwalająca na wyszukiwanie problemów związanych z wydajnością. Analiza informacji zebranych po ustawieniu parametru `log_statement` i powiązanych źródeł informacji szczegółowych na poziomie poleceń może ukazać prawdziwą przyczynę wielu rodzajów problemów wydajności. Zebrane w ten sposób informacje można połączyć z odpowiednimi narzędziami analizy, które omówiono w rozdziale 7.

log_min_duration_statement

Gdy znamy już ilość czasu wymaganą do wykonania typowego zapytania, użycie parametru `log_min_duration_statement` pozwoli na rejestrowanie tylko tych poleceń, których wykonanie przekroczy ustalony czas. Wartość jest podawana w milisekundach, tak więc parametr można ustawić w następujący sposób:

```
log_min_duration_statement=1000
```

Powyższe polecenie powoduje rejestrację w dzienniku zdarzeń tylko poleceń wykonywanych dłużej niż sekundę. Parametr może być użyteczny do wyszukiwania źródeł „odstających" poleceń, których wykonanie trwa znacznie dłużej niż pozostałych.

Jeżeli używany jest PostgreSQL w wersji 8.4 lub nowszej, zamiast tej funkcji preferowanym rozwiązaniem może być moduł `auto_explain`. Więcej informacji na temat modułu można znaleźć na stronie *http://www.postgresql.org/docs/8.4/static/auto-explain.html*. Moduł umożliwia sprawdzenie operacji wykonywanych przez powolne zapytania po przejrzeniu powiązanych z nimi planów `EXPLAIN`.

Polecenie VACUUM i dane statystyczne

Baza danych PostgreSQL wymaga dwóch podstawowych form regularnej obsługi podczas dodawania, uaktualniania i usuwania danych.

Polecenie `VACUUM` powoduje usunięcie śmieci po starych transakcjach, łącznie z usunięciem informacji, które nie są dłużej widoczne, oraz zwrócenie zwolnionego miejsca, tak aby mogło być ponownie wykorzystane. Im więcej w bazie danych wykonywanych poleceń `UPDATE` i `DELETE`, tym częściej trzeba przeprowadzać operację usuwania śmieci. Nawet w statycznych tabelach, których zawartość nigdy nie ulega zmianie, od czasu do czasu trzeba wykonać operację usuwania śmieci.

Polecenie `ANALYZE` przegląda tabele w bazie danych i zbiera na ich temat dane statystyczne — informacje, takie jak liczba posiadanych rekordów bądź liczba odmiennych wartości w tabelach. Wiele aspektów planowania zapytania zależy od prawidłowego zebrania tych danych.

Więcej informacji na temat polecenia `VACUUM` znajduje się w rozdziale 7., natomiast temat stosowania danych statystycznych poruszono w rozdziale 10.

Demon autovacuum

Ponieważ zadanie usuwania śmieci ma w dłuższej perspektywie znaczenie krytyczne dla bazy danych, od wydania PostgreSQL 8.1 w serwerze umieszczono demon `autovacuum`, który działa w tle i zajmuje się wspomnianym zadaniem. Aktywacja demona następuje po wykonaniu w bazie danych liczby zmian przekraczającej wartość obliczoną na podstawie wielkości tabeli.

Parametr dla demona autovacuum jest włączony domyślnie w PostgreSQL 8.3, a wartość domyślna jest ustalona w sposób wystarczający do działania mniejszej bazy danych i wymaga jedynie niewielkiej ręcznej modyfikacji. Ogólnie rzecz biorąc, trzeba uważać, aby ilość danych w obszarze FSM nie przekraczała wartości ustalonej przez parametr max_fsm_pages. Od PostgreSQL 8.4 opisane wymaganie nie stanowi już powodu do zmartwień dla użytkownika.

Włączanie demona autovacuum w starszych wersjach

Jeżeli demon autovacuum jest dostępny, ale nie jest włączony domyślnie, jak ma to miejsce w bazach danych PostgreSQL 8.1 i 8.2, istnieje kilka powiązanych z nim parametrów, które muszą być włączone w celu zapewnienia jego prawidłowej pracy. Więcej informacji na ten temat przedstawiono na stronach *http://www.postgresql.org/docs/8.1/interactive/maintenance.html* i *http://www.postgresql.org/docs/8.2/interactive/routine-vacuuming.html*.

W wymienionych wersjach w pliku konfiguracyjnym *postgresql.conf* trzeba ustawić trzy parametry:

```
stats_start_collector=true
stats_row_level=true
autovacuum=on
```

Warto zwrócić uwagę, że — zgodnie z ostrzeżeniem przedstawionym w dokumentacji — zalecane jest również użycie parametru superuser_reserved_connections zezwalającego na działanie demonów autovacuum w wymienionych wersjach bazy danych.

Demon autovacuum dostępny w wersjach 8.1 i 8.2 nie jest aż tak wydajny jak dostarczany w wersjach 8.3 i nowszych. Uzyskanie odpowiedniego balansu zapewniającego prawidłową obsługę bazy danych bez zbyt dużego obciążenia serwera będzie wymagało pewnych eksperymentów i dopasowywania wartości. Ponieważ jest to tylko jeden proces, pozostawienie go, by działał w tle, gdy serwer jest zajęty, będzie mniejszym obciążeniem dla serwera. Zagadnienie to nie będzie tutaj obszernie omawiane. Ogólnie rzecz biorąc, znacznie lepszym rozwiązaniem jest poświęcenie czasu na uaktualnienie bazy danych PostgreSQL do wydania zawierającego nowszą wersję demona autovacuum niż próba dostosowania starszej wersji. Szczególnie dotyczy to sytuacji, kiedy równocześnie występują inne problemy z wydajnością, których nie można tak łatwo rozwiązać w starszej wersji serwera.

maintenance_work_mem

Kilka operacji w serwerze bazy danych wymaga pamięci roboczej dla operacji większych niż zwykłe sortowanie. Polecenia VACUUM, CREATE INDEX oraz ALTER TABLE ADD FOREIGN KEY mogą alokować maksymalną ilość pamięci określoną przez parametr maintenance_work_mem. Jest mało prawdopodobne, aby wiele sesji przeprowadzało jednocześnie tego rodzaju operacje. Dla tego parametru można określić wartość znacznie wyższą niż w standardowej opcji work_mem ustawianej dla każdego klienta. Warto pamiętać, że przynajmniej parametr autovacuum_max_workers

(dla którego od wersji 8.3 wartość domyślna wynosi 3) zaalokuje tę ilość pamięci, dlatego też podczas określania tej wartości należy uważać na takie sesje (prawdopodobnie z jedną lub dwiema sesjami wykonującymi operację `CREATE INDEX`).

Przy założeniu, że liczba demonów `autovacuum` nie została zwiększona, typowe wysokie ustawienie omawianej wartości w nowoczesnym serwerze będzie miało wartość 5% całkowitej ilości pamięci RAM. Tak więc nawet pięć procesów nie wykorzysta więcej niż jedną czwartą dostępnej pamięci RAM. Oznacza to, że `maintenance_work_mem` wykorzystuje około 50 MB pamięci na każdy dostępny gigabajt pamięci RAM w serwerze.

default_statistics_target

PostgreSQL podejmuje decyzje dotyczące sposobu wykonywania zapytań na podstawie danych statystycznych zebranych dla każdej tabeli i przechowywanych w bazie danych. Tego rodzaju informacje są zbierane podczas analizy tabel za pomocą polecenia `ANALYZE` lub demona `autovacuum`. W każdym przypadku ilość informacji zbierana w trakcie analizy jest określana przez parametr `default_statistics_target`. Zwiększenie wartości tego parametru wydłuża analizę. Analiza regularnie przeprowadzana przez demon `autovacuum` w tle stanowi coraz większe obciążenie podczas konserwacji bazy danych. Jednak brak odpowiednich danych statystycznych dotyczących tabeli skutkuje ryzykiem użycia niewłaściwego planu wykonywania zapytań względem tej tabeli.

Wartość domyślna wymienionej opcji z reguły jest bardzo niska (to znaczy 10), ale w PostgreSQL 8.4 została zwiększona do 100. Stosowanie większej wartości było popularne także w starszych wersjach bazy danych w celu ogólnego poprawienia zachowania zapytań. Indeksy korzystające z operatora `LIKE` działają znacznie lepiej, gdy opisywany parametr ma wartość wyższą niż 100, a nie niższą, co wiąże się ze zdefiniowaną na stałe zmianą sposobu ich działania po użyciu wartości większej niż 100 dla tego parametru.

Warto zwrócić uwagę, że zwiększenie wartości parametru `default_statistics_target` skutkuje ogólnym spowolnieniem systemu, jeżeli w ogóle nie są wykonywane zapytania, w których dodatkowe dane statystyczne powodują wybór lepszego planu wykonywania. Z tego powodu pewne proste testy wydajności pokazują, że baza danych PostgreSQL w wersji 8.4 jest nieco wolniejsza od 8.3, jeżeli obie stosują wartości domyślne parametrów. W niektórych przypadkach po instalacji wersji 8.4 można zmniejszyć wartość wymienionego parametru. Ustawianie wyjątkowo dużej wartości parametru `default_statistics_target` jest odradzane, ze względu na generowane wówczas ogromne obciążenie dla bazy danych.

Jeżeli w tabeli znajduje się kolumna, o której wiadomo, że potrzebuje lepszych danych statystycznych, można względem tej kolumny wykonać polecenie `ALTER TABLE SET STATISTICS` w celu dostosowania dla niej wartości parametru. Takie rozwiązanie jest lepsze niż zwiększanie wartości domyślnej parametru dla całego systemu, gdyż wtedy każda tabela „płaci" za wymaganie jednej kolumny w bazie danych. Zazwyczaj kolumny wymagające do prawidłowej pracy naprawdę dużej ilości danych statystycznych muszą mieć wartość parametru `default_statistics_target` równą 1000 (w nowszych wersjach PostgreSQL zwiększono ją do 10000), aby zmiana była

widoczna. Jest to wartość dużo większa niż ilość danych statystycznych konieczna do zebrania w każdej tabeli bazy danych.

Punkty kontrolne

Sposób działania mechanizmu punktów kontrolnych i dotyczące go parametry przedstawiono w poprzednim rozdziale. W tym miejscu nacisk położono na powszechnie stosowane praktyki ustawiania wartości początkowych dla tych parametrów.

checkpoint_segments

Każdy segment mechanizmu WAL zabiera do 16 MB. Jak przedstawiono na stronie *http://www.postgresql.org/docs/current/interactive/wal-configuration.html*, maksymalną liczbę segmentów, która może być w użyciu w danym momencie, można obliczyć ze wzoru:

```
(2 + checkpoint_completion_target) * checkpoint_segments + 1
```

Warto zwrócić uwagę, że baza danych PostgreSQL w wersjach wcześniejszych niż 8.3 nie obsługuje rozproszonych punktów kontrolnych, ale nadal można użyć powyższego wzoru i po prostu dla brakującej wartości wykorzystać poniższy fragment kodu:

```
checkpoint_completion_target=0
```

Wynik to całkowita wielkość wszystkich segmentów mechanizmu WAL, które mogą znaleźć się na dysku. Pozwala to zarówno na ustalenie ilości zajmowanego miejsca na dysku twardym, jak i oszacowanie ilości czasu potrzebnego na przeprowadzenie naprawy po awarii bazy danych. Oczekiwany wzrost wielkości dziennika zdarzeń `pg_xlog` został przedstawiony w poniższej tabeli.

checkpoint_segments	checkpoint_completion_target=0	target=0.5	target=0.9
3	112 MB	144 MB	160 MB
10	336 MB	416 MB	480 MB
32	1040 MB	1296 MB	1504 MB
64	2064 MB	2576 MB	2992 MB
128	4112 MB	5136 MB	5968 MB
256	8208 MB	10256 MB	11904 MB

Ogólny wniosek, który można wyciągnąć na podstawie przedstawionych tutaj danych, jest następujący: na każde 32 segmenty punktów kontrolnych trzeba się liczyć z akumulacją około 1 GB danych mechanizmu WAL. Ponieważ w przypadku takiej ilości danych naprawa bazy danych po awarii może zająć sporo czasu, wartość 32 to największa, jaką warto zastosować w poważnym, produkcyjnym serwerze bazy danych. Wartość domyślna 3 jest jednak dla większości systemów stanowczo za mała, nawet w niewielkich instalacjach należy rozważyć jej zwiększenie do 10.

Wartość większą niż 32 stosuje się zazwyczaj tylko w mniejszych serwerach przeprowadzających operacje typu *bulk-loading*, ponieważ może to znacznie zwiększyć wydajność, a czas naprawy bazy danych po ewentualnej awarii jest nieistotny. Bazy danych, które regularnie przeprowadzają operacje *bulk-loading*, mogą wymagać większej wartości parametru `checkpoint_segments`.

checkpoint_timeout

Wartość domyślna tego parametru wynosząca 5 minut jest wystarczająca dla większości instalacji. Jeżeli system nie nadąża z operacjami zapisu i zwiększono wartość parametru `checkpoint_segments` w taki sposób, że parametr `checkpoint_timeout` stał się podstawowym sposobem inicjowania operacji tworzenia punktu kontrolnego, rozsądne będzie rozważenie zwiększenia wartości także dla parametru `checkpoint_timeout`. Ustalenie przerwy w długości dziesięciu minut lub nawet więcej między tworzeniem punktów kontrolnych nie jest niebezpieczne, po prostu wydłuża czas naprawy bazy danych po ewentualnej awarii. Ponieważ jest to jeden z parametrów, które mają wpływ na długość czasu niedostępności bazy danych po awarii, jego wartość należy ustawiać bardzo ostrożnie.

checkpoint_completion_target

Po zwiększeniu wartości parametru `checkpoint_segments` do przynajmniej 10 rozsądne wydaje się również zwiększenie wartości parametru `checkpoint_competion_target` do jego praktycznego maksimum wynoszącego 0,9. W ten sposób uzyskiwane jest maksymalne rozproszenie punktów kontrolnych, co przynajmniej teoretycznie oznacza mniejsze obciążenie związane z operacjami wejścia-wyjścia. Jednak w niektórych sytuacjach utrzymywanie wartości domyślnej 0,5 nadal będzie lepszym rozwiązaniem, ponieważ zmniejsza prawdopodobieństwo, że operacje zapisu jednego punktu kontrolnego zazębią się z operacjami drugiego.

Wydaje się nieprawdopodobne, aby wartość poniżej 0,5 była w ogóle efektywna dla rozpraszania punktów kontrolnych. Co więcej, o ile liczba segmentów nie jest naprawdę ogromna, małe zmiany wartości parametru mają niewielkie znaczenie. Jedyne sensowne podejście polega na wypróbowaniu obu wartości (0,5 i 0,9) w aplikacji i sprawdzeniu na poziomie monitorowania systemu operacyjnego, która powoduje mniejsze obciążenie związane z operacjami wejścia-wyjścia.

Ustawienia mechanizmu WAL

Mechanizm WAL (ang. *Write-Ahead Log*) używany w PostgreSQL został omówiony w rozdziale 5.

wal_buffers

Dokumentacja parametru `wal_buffers` sugeruje, że wartość domyślna wynosząca 64 kB jest wystarczająca tak długo, dopóki pojedyncza transakcja nie przekracza wymienionej wartości.

Jednak w praktyce testy wydajności intensywnie wykorzystujące operacje zapisu pokazują, że optymalną wydajność można osiągnąć po ustawieniu znacznie większej wartości, co najmniej 1 MB lub więcej. Jedyną wadą jest zwiększony poziom wykorzystania pamięci współdzielonej. Biorąc po uwagę, że nie ma potrzeby, aby więcej niż pojedynczy segment WAL musiał być buforowany, a także ilość pamięci instalowaną w nowoczesnych serwerach, parametrowi `wal_buffers` można obecnie przydzielić znacznie większą wartość:

```
wal_buffers=16MB
```

Po ustawieniu powyższej wartości można zapomnieć o parametrze `wal_buffers` jako potencjalnym wąskim gardle lub komponencie, który później trzeba optymalizować. W przypadku niewielkiej ilości pamięci w serwerze można rozważyć użycie mniejszej wartości.

wal_sync_method

W rozdziale 2. wyjaśniono, jak ważne jest skonfigurowanie serwera w taki sposób, aby unikał wykorzystywania nietrwałych buforów zapisu. Jednym z celów parametru `wal_sync_method` jest optymalizacja zachowania podczas korzystania z nietrwałych buforów zapisu.

Zachowanie domyślne jest w pewien sposób odmienne od większości opcji. Podczas kompilacji kodu źródłowego serwera rozważane są różne możliwe sposoby zapisu. Jeden, uznany za najbardziej efektywny, staje się później stosowany domyślnie w trakcie kompilacji. Jednak wartość ta nie jest zapisana w pliku konfiguracyjnym *postgresql.conf* wygenerowanym przez `initdb`, co odróżnia ją od innych automatycznie wykrywanych i charakterystycznych dla platform wartości, takich jak `shared_buffers`.

Przed zmianą czegokolwiek należy za pomocą polecenia `SHOW` sprawdzić, co w używanej przez czytelnika platformie zostało wykryte jako najszybsza, bezpieczna metoda. W systemie Linux wynik jest następujący:

```
postgres=# show wal_sync_method;
 wal_sync_method
-----------------
 fdatasync
```

Z kolei na platformach Windows i Mac OS X istnieje specjalny parametr gwarantujący, że system operacyjny wyczyści wszystkie bufory zapisu. Na wspomnianych platformach wartością bezpieczną, która włącza takie zachowanie, jest:

```
wal_sync_method=fsync_writethrough
```

Jeżeli ustawienie takie jest dostępne, naprawdę należy je wykorzystać! Pracuje ono prawidłowo i gwarantuje wykonywanie bezpiecznych operacji zapisu w bazie danych bez spowalniania innych aplikacji, co ma miejsce w przypadku całkowitego wyłączenia bufora zapisu w dysku twardym.

Jednak powyższe ustawienie nie będzie funkcjonowało na wszystkich platformach. Warto pamiętać, że po zmianie wartości domyślnej na przedstawioną powyżej nastąpi pewien spadek wydajności. Dzieje się tak zawsze podczas przejścia z buforowania niebezpiecznego do bezpiecznego.

Na innych platformach optymalizacja parametru `wal_sync_method` może być znacznie bardziej skomplikowana. Teoretycznie istnieje możliwość poprawienia wydajności operacji zapisu w systemach z rodziny Unix poprzez zmianę dowolnej metody zapisu używającej par `write/fsync` lub `write/fdatasync` na stosującą zapis prawdziwie synchroniczny. Można się przekonać, po wydaniu polecenia `SHOW`, że na platformach obsługujących bezpieczne zachowanie zapisu `DSYNC` jest to opcja stosowana domyślnie:

```
wal_sync_method=open_datasync
```

Jednak także w tym przypadku opcja ta nie jest jawnie podana w pliku konfiguracyjnym. Jeżeli tak jest na używanej przez czytelnika platformie, można przeprowadzić jedynie niewielką optymalizację. Wartość `open_datasync` to ogólnie optymalne podejście, a kiedy wymieniona opcja jest dostępna, pozwala nawet na bezpośrednie używanie urządzeń wejścia-wyjścia, a także pomijanie bufora systemu operacyjnego.

W systemie Linux sytuacja jest prawdopodobnie najbardziej skomplikowana. Jak przedstawiono we wcześniejszym fragmencie kodu, platforma Linux domyślnie używa metody `fdatasync`. Za pomocą poniższego polecenia można włączyć tryb synchronicznych operacji zapisu:

```
wal_sync_method=open_sync
```

Ponadto w wielu przypadkach można odkryć, że metoda ta działa szybciej — czasami nawet znacznie szybciej — niż określona domyślnie. Jednak bezpieczeństwo jej stosowania zależy od używanego systemu plików. W większości systemów Linux domyślnym systemem plików jest ext3, który w wielu przypadkach nie obsługuje bezpiecznych operacji zapisu `O_SYNC`, co może doprowadzić do uszkodzenia danych. Przykłady niebezpieczeństw takiego ustawienia na platformie Linux przedstawiono w wątku „PANIC caused by open_sync on Linux" znajdującym się na stronie *http://archives.postgresql.org/pgsql-hackers/2007-10/msg01310.php*. Istnieją dowody, że w ostatnich wersjach jądra (2.6.32) ten problem nie istnieje, jeśli używany jest system plików ext4, ale takie rozwiązanie nie zostało jeszcze intensywnie przetestowane na poziomie bazy danych.

W każdym przypadku podczas przeprowadzania własnych testów wartości parametru `wal_sync_method` należy wykonać również test polegający na „wyciągnięciu z gniazda sieciowego wtyczki przewodu zasilającego serwer". Spowoduje to nagłe odcięcie zasilania serwera i pozwoli na sprawdzenie, czy nastąpiła utrata jakichkolwiek danych w wyniku stosowania testowanej metody. Zalecane jest również przeprowadzenie długotrwałych testów przy dużym obciążeniu, aby znaleźć sporadyczne błędy, które mogą doprowadzić do awarii.

Replikacja WAL i PITR

Parametry `archive_mode`, `archive_command` oraz `archive_timeout` zostały omówione w rozdziale 14.

Ustawienia na poziomie klienta

Wprawdzie wszystkie parametry omawiane w tym podrozdziale mogą być dostosowane na poziomie klienta, pożądane jest ustawienie dla nich odpowiednich wartości początkowych w głównym pliku konfiguracyjnym. Gdy trzeba zmienić wartość parametru w poszczególnych klientach, zawsze można ją wprowadzić na czas trwania sesji, wykorzystując do tego polecenie `SET`.

effective_cache_size

Jak wspomniano w poprzednim rozdziale, baza danych PostgreSQL używa zarówno własnej dedykowanej jej pamięci (`shared_buffers`), jak i korzysta z bufora systemu plików. W pewnych sytuacjach, podczas podejmowania decyzji dotyczących efektywności użycia indeksu bądź nie, baza danych porównuje obliczone przez siebie wielkości względem efektywnej sumy wszystkich wymienionych buforów. Wartość tę spodziewa się znaleźć w parametrze `effective_cache_size`.

Ta sama reguła, wedle której wartość parametru `shared_buffers` powinna wynosić około 25% ilości pamięci systemowej, mówi także, że wartość parametru `effective_cache_size` powinna wynosić od 50 do 75% ilości pamięci RAM. Aby znacznie dokładniej oszacować wartości, trzeba na początek przyjrzeć się buforowi systemu plików:

- w systemach z rodziny Unix oszacowanie wielkości bufora systemu plików wymaga dodania wartości `free` i `cached` wyświetlanych przez polecenia `free` i `top`;
- w systemach Windows należy wyświetlić okno Menedżera zadań Windows, przejść na zakładkę *Wydajność* i odczytać wartość w linii *Bufor systemu*.

Po przyjęciu, że baza danych została już uruchomiona, do obliczonych wartości trzeba dodać jeszcze wartość `shared_buffers`, w ten sposób otrzyma się `effective_cache_size`. Jeżeli baza danych nie została jeszcze uruchomiona, bufor systemu operacyjnego będzie zwykle wystarczająco dokładny do oszacowania wartości parametru. Po uruchomieniu bazy danych większość pamięci przeznaczonej dla bazy danych zwykle i tak będzie zaalokowana na potrzeby jej bufora.

Parametr `effective_cache_size` nie powoduje alokacji jakiejkolwiek pamięci. Jest używany wyłącznie po to, by określić sposób wykonywania zapytań, i jego ogólne oszacowanie jest wystarczające do większości celów. Kiedy jednak wartość ta będzie zbyt wysoka, wykonywanie zapytań może skutkować zakłóceniami w buforze zarówno bazy danych, jak i systemu operacyjnego

wynikającymi z konieczności odczytania ogromnej liczby bloków wymaganych do wykonania zapytania, które bardzo łatwo zmieści się w pamięci RAM.

Bardzo rzadko zdarza się, aby parametr ten był optymalizowany na poziomie klienta, nawet jeśli jest to możliwe.

synchronous_commit

W rozdziale 2. obciążenie związane z oczekiwaniem, aż fizyczny dysk zakończy operację zatwierdzania, wskazano jako potencjalne wąskie gardło podczas zatwierdzania transakcji. Jeżeli czytelnik nie posiada podtrzymywanego bateryjnie bufora zapisu, który mógłby przyśpieszyć takie operacje, ale koniecznie chce przyśpieszyć operacje zatwierdzania transakcji, rodzi się pytanie, w jaki sposób można to zrobić. Standardowym podejściem jest wyłączenie opcji `synchronous_commit`, która czasami nazywana jest opcją włączającą asynchroniczne zatwierdzenia. Powoduje ona zgromadzenie w większą grupę operacji zatwierdzenia w częstotliwości określonej przez powiązany z nią parametr `wal_writer_delay`. Ustawienia domyślne gwarantują realne zatwierdzenie transakcji na dysku po upływie co najwyżej 600 milisekund od chwili zatwierdzenia transakcji przez klienta. W trakcie tego okresu czasu, który można skrócić, licząc się jednocześnie ze spadkiem szybkości, niezatwierdzone dane nie będą odzyskane po ewentualnej awarii serwera.

Warto zwrócić uwagę, że opisywany parametr można wyłączyć dla pojedynczego klienta na czas trwania jego sesji, zamiast wyłączać go dla całego serwera. Wyłączenie dla klienta odbywa się za pomocą polecenia `SET`:

```
SET LOCAL synchronous_commit TO OFF;
```

W ten sposób czytelnik dysponuje opcją stosowania różnych fizycznych gwarancji zatwierdzania dla odmiennych typów danych wstawianych do bazy danych. Aktywnie monitorowana tabela — do której dane są najczęściej wstawiane, a ułamek sekundy strat akceptowany — jest dobrym kandydatem dla asynchronicznych zatwierdzeń. W przypadku rzadziej zapisywanej tabeli przechowującej rzeczywiste transakcje finansowe preferowanym rozwiązaniem jest stosowanie standardowych, synchronicznych zatwierdzeń.

work_mem

Kiedy wykonywane jest zapytanie wymagające sortowania danych, baza danych szacuje ilość danych koniecznych do użycia i następnie porównuje obliczoną wartość do określonej w parametrze `work_mem`. Jeżeli jest większa (a wartość domyślna parametru `work_mem` wynosi 1 MB), zamiast sortować dane w pamięci, zapisze je na dysku i przeprowadzi operację sortowania dyskowego. Taka operacja jest znacznie wolniejsza niż sortowanie przeprowadzane w pamięci. Dlatego też, w przypadku regularnego sortowania danych i posiadania wolnej pamięci, zwiększenie wartości parametru `work_mem` może być jednym z najefektywniejszych sposobów przyśpieszenia działania serwera. Magazyny danych generujące ogromne raporty mają w swoich serwerach gigabajty pamięci przypisane parametrowi `work_mem`.

Problem polega na tym, że niekoniecznie można łatwo przewidzieć liczbę operacji sortowania przeprowadzanych przez klienta, a parametr `work_mem` jest określany dla każdej operacji sortowania, a nie dla każdego klienta. Oznacza to, że ilość pamięci używanej przez `work_mem` może być teoretycznie nieograniczona, jeżeli liczba klientów jednocześnie przeprowadzających sortowanie będzie wystarczająco duża.

W praktyce w typowym zapytaniu nie ma zbyt wielu operacji sortowania, najczęściej tylko jedna bądź dwie. Ponadto nie wszyscy aktywni klienci będą w tym samym czasie przeprowadzali operacje sortowania. W trakcie określania ilości pamięci dla parametru `work_mem` zwykle bierze się pod uwagę ilość wolnej pamięci RAM po alokacji `shared_buffers` (ta sama wielkość bufora systemu operacyjnego obliczana na potrzeby parametru `effective_cache_size`), dzieli ją przez wartość `max_connections`, a następnie bierze tylko część obliczonej wartości. Połowa obliczonej liczby będzie wartością dość dużą dla parametru `work_mem`. W takim przypadku w serwerze może zabraknąć pamięci, jeżeli każdy klient będzie przez cały czas przeprowadzał po dwie operacje sortowania, ale prawdopodobieństwo wystąpienia takiej sytuacji jest znikome.

Obliczenia wartości `work_mem` są coraz częściej stosowane w najnowszych wersjach PostgreSQL w celu oszacowania, czy struktura hash może zostać zbudowana w pamięci. Pamięć jest używana także przez klienta, a nie jedynie przeznaczona dla operacji sortowania. Przedstawiono po prostu najłatwiejszy sposób rozwiązywania problemów dotyczących rodzajów alokacji pamięci.

Podobnie jak `synchronous_commit`, parametr `work_mem` może być również ustawiany na poziomie klienta. W ten sposób wartość domyślną można zdefiniować na średnim poziomie i zwiększać pamięć do sortowania dla użytkowników, którzy wykonują zapytania generujące ogromne raporty.

random_page_cost

Parametr często jest optymalizowany, ale jego objaśnienie wymaga przedstawienia wielu informacji dotyczących planowania zapytań. Temat ten został omówiony w rozdziale 10. Szczególnie we wcześniejszych wersjach PostgreSQL zmniejszenie wartości parametru `random_page_cost` poniżej wartości domyślnej — na przykład z 4.0 na 2.0 — było powszechnie stosowanym rozwiązaniem. Celem było zwiększenie prawdopodobieństwa, że planista zapytania wykorzysta zapytania indeksowane zamiast alternatywy w postaci skanowania sekwencyjnego. Ponieważ w nowszych wersjach serwera wbudowany jest sprytniejszy planista zapytania, wspomnianej techniki nie należy stosować. O wiele rozsądniejszym rozwiązaniem jest zebranie lepszych danych statystycznych i wykorzystanie parametrów dotyczących pamięci jako podstawowych sposobów wpływania na planistę zapytań.

constraint_exclusion

Jeżeli używana jest baza danych PostgreSQL w wersji 8.3 lub wcześniejszej, a do partycjonowania danych zastosowano oferowaną przez bazę danych funkcję tabeli dziedziczenia, parametr `constraint_exclusion` musi być włączony. Powody takiego stanu rzeczy wyjaśniono w rozdziale 15.

Począwszy od wersji 8.4, wartością domyślną parametru `constraint_exclusion` jest nowe, sprytniejsze ustawienie o nazwie `partition`, które w większości przypadków sprawdza się doskonale i nie musi być modyfikowane.

Optymalizacje niezalecane

W pliku konfiguracyjnym *postgresql.conf* znajduje się kilka parametrów, dla których w innych poradnikach przedstawiono kiepskie wskazówki lub w serwerze administrowanym przez czytelnika mają ustawione nieprawidłowe wartości. Inne mają nazwy sugerujące użycie wraz z parametrami, które w rzeczywistości nie istnieją. W tym punkcie opisano najczęściej spotykane opcje, których optymalizacja jest niezalecana.

fsync

Jeżeli czytelnik chce zignorować proces naprawy po wystąpieniu awarii, może to zrobić poprzez wyłączenie parametru `fsync`. W ten sposób wartość parametru `wal_sync_method` nie będzie miała znaczenia, ponieważ serwer i tak nie wykona żadnych wywołań `sync` mechanizmu WAL.

Trzeba w tym miejscu zdać sobie sprawę, że w przypadku wystąpienia jakiejkolwiek awarii przy wyłączonym parametrze `fsync` baza danych prawdopodobnie będzie uszkodzona i uruchomienie serwera stanie się niemożliwe. Wprawdzie to okropna sytuacja, ale zwiększenie wydajności w wyniku wyłączenia procesu naprawy po awarii jest tak ogromne, że czytelnik może natknąć się na sugestie wyłączenia parametru `fsync`. Równie nieufnie należy traktować inne rady pochodzące z tych samych źródeł, które sugerują wyłączenie parametru `fsync`, ponieważ wyłączenie tego parametru jest szalenie niebezpieczne.

Jedynym powodem, dla którego taki pomysł zyskał zwolenników we wcześniejszych wersjach PostgreSQL, był brak innego sposobu na zmniejszenie liczby wywołań `fsync` — to taki kompromis: nieco większa wydajność kosztem mniejszej niezawodności. Od wersji 8.3 użytkownicy, zamiast wyłączać parametr `fsync`, zrobią lepiej, wyłączając parametr `synchronous_commit`.

Istnieje jedna sytuacja, gdy użycie parametru `fsync=off` nadal ma sens — to początkowe operacje typu *bulk-loading*. Jeżeli do bazy wstawiane są ogromne ilości danych, a sprzęt nie jest wyposażony w bateryjnie podtrzymywany bufor zapisu, taka operacja wstawiania danych będzie trwała zdecydowanie za długo, aby uznać ją za praktyczną. W takim przypadku wyłączenie parametru `fsync` podczas przeprowadzania operacji wstawiania danych — w przypadku awarii serwera wszystkie dane i tak można łatwo odtworzyć — może być jedynym sposobem przyśpieszenia całej operacji. Po zakończeniu procesu wstawiania należy z powrotem włączyć parametr `fsync`.

W niektórych systemach parametr `fsync` jest wyłączany w serwerach posiadających redundancyjną kopię bazy danych — na przykład w systemach zapasowych używanych do generowania raportów. Jeżeli w takim przypadku dane zostaną uszkodzone, zawsze można przeprowadzić ponowną synchronizację z głównym systemem.

full_page_writes

Podobnie jak przy `fsync`, wyłączenie parametru `full_page_writes` zwiększa wydajność za cenę większego ryzyka uszkodzenia bazy danych. Przed wyłączeniem tego parametru należy bardzo dokładnie i ostrożnie przeanalizować możliwości systemu plików oraz używanego sprzętu, aby mieć gwarancję, że nie dojdzie do częściowego zapisu stron.

commit_delay i commit_siblings

Przed implementacją `synchronous_commit` podejmowane były próby dodania tego rodzaju funkcji za pomocą parametrów `commit_delay` i `commit_siblings`. W większości przypadków nie są to parametry efektywne do optymalizacji. Uzyskanie realnego przyśpieszenia poprzez dostosowanie wartości tych parametrów jest niezwykle trudne, za to bardzo łatwo można doprowadzić do spowolnienia wykonywania każdej transakcji. Jedyny przypadek, kiedy opisywane parametry mogą okazać się przydatne, dotyczy systemów o ogromnej ilości operacji wejścia-wyjścia. Ustawienie bardzo małej wartości opóźnienia może spowodować przeprowadzanie operacji zapisu w większych blokach, co czasami w połączeniu z większą wartością jednostki macierzy RAID okazuje się lepszym rozwiązaniem.

max_prepared_transactions

Wielu użytkowników po spojrzeniu na nazwę tego parametru będzie przekonanych, że dotyczy transakcji składowanych, czyli techniki powszechnie stosowanej w celu uniknięcia wstawiania złośliwego kodu SQL, i odczuje potrzebę zwiększenia wartości tego parametru. To jednak błędne założenie, omawiany parametr i transakcje składowane nie są ze sobą powiązane. Transakcja składowana to taka, która używa polecenia `PREPARE TRANSACTION` w celu użycia dwuetapowego zatwierdzenia (ang. *two-phase commit*, 2PC). Jeżeli czytelnik nie używa tego polecenia oraz techniki 2PC, może pozostawić dla tego parametru wartość domyślną. Natomiast w przypadku używania funkcji dwuetapowego zatwierdzania konieczność zwiększenia wartości parametru `max_prepared_transaction` wystąpi prawdopodobnie tylko wtedy, kiedy trzeba będzie dopasować ją do liczby połączeń.

Zapytania włączające parametry

Istnieje możliwość wyłączenia wielu technik stosowanych przez planistę zapytania, co ma na celu uniknięcie użycia niewłaściwego typu zapytania. Czasami jest to stosowane jako rodzaj obejścia problemu polegającego na tym, że PostgreSQL nie obsługuje bezpośrednich podpowiedzi optymalizatora dotyczących sposobu wykonania zapytania. Czytelnik mógł się spotkać z przedstawionym poniżej wierszem kodu jako sugerowanym sposobem wymuszenia użycia indeksów zamiast skanowania sekwencyjnego:

```
enable_seqscan = off
```

Ogólnie rzecz ujmując, to zła praktyka. Zamiast niej, aby podejmować lepsze decyzje, należy dążyć do poprawienia informacji przekazywanych optymalizatorowi zapytań. Temat został omówiony w rozdziale 10.

Optymalizacja ustawień nowego serwera

Istnieje kilka sposób połączenia wszystkich przedstawionych dotąd informacji w jeden proces optymalizacji ustawień nowego serwera. Wybór najlepszego sposobu zależy od oczekiwań stawianych serwerowi, a także od samego użytkownika — czy chce przeprowadzić samodzielną optymalizację, czy będzie stosował zalecane tutaj wartości parametrów.

Wskazówki dotyczące serwerów dedykowanych

Początkową optymalizację serwera można sprowadzić do całkiem mechanicznego procesu. Oto on.

1. Dostosowanie rejestracji zdarzeń w taki sposób, aby dostarczała większej ilości informacji.
2. Ustalenie wielkości parametru `shared_buffers`. Należy rozpocząć od wartości równej 25% pamięci systemowej. Jeżeli czytelnik używa najnowszych wersji PostgreSQL obsługujących rozpraszanie punktów kontrolnych i wie, że dane obciążenie serwera odniesie korzyści po przydzieleniu buforowi większej ilości pamięci, może spróbować zwiększania wartości `shared_buffers`. W przypadku platformy, na której parametr ten nie jest tak użyteczny, jego wartość należy odpowiednio dopasować lub nawet zmniejszyć.
3. Rozsądne oszacowanie maksymalnej liczby połączeń. Jest to ograniczenie bezwzględne. Po osiągnięciu ustalonej liczby połączeń próby nawiązania połączenia przez nowych klientów będą odrzucane.
4. Uruchomienie serwera z tymi parametrami początkowymi. Następnie trzeba sprawdzić ilość pamięci pozostałej dla bufora systemu plików.
5. Kolejny krok to określenie wielkości parametru `effective_cache_size` na podstawie wielkości parametru `shared_buffers` i bufora systemu operacyjnego.
6. Podzielenie wielkości bufora systemu operacyjnego przez wartość `max_connections`, a później jeszcze przez dwa. W ten sposób czytelnik otrzymuje rozsądną wartość dla parametru `work_mem`. Jeżeli aplikacja w żaden sposób nie jest uzależniona od wydajności operacji sortowania, odpowiednie będzie ustawienie znacznie niższej wartości parametru `work_mem`.
7. Ustawienie dla parametru `maintenance_work_mem` wartości około 50 MB na każdy gigabajt pamięci RAM.
8. Zwiększenie wartości parametru `checkpoint_segments` do przynajmniej 10. Jeżeli czytelnik posiada sprzęt klasy serwerowej z bateryjnie podtrzymywanym buforem zapisu, wartość 32 dla parametru `checkpoint_segments` będzie znacznie lepszą wartością domyślną.

9. Zmiana wartości domyślnej parametru `wal_sync_method`. Jeżeli używana jest platforma, na której wartość domyślna tego parametru jest niebezpieczna, należy zmienić tę wartość.
10. Wartość parametru `wal_buffers` trzeba zwiększyć do 16 MB.
11. Dla baz danych PostgreSQL w wersjach niższych niż 8.4 warto rozważyć zwiększenie wartości parametrów `default_statistics_target` (do 100, wartość domyślna w nowszych wersjach PostgreSQL) i `max_fsm_pages` na podstawie informacji o przewidywanym obciążeniu dla tej bazy danych.

Po skonfigurowaniu pewnej liczby serwerów przeznaczonych do obsługi danej aplikacji czytelnik będzie miał znacznie lepsze rozeznanie, które wartości początkowe mają sens, aby rozpoczynać z nimi pracę. Szczególnie wartości końcowe parametrów `checkpoint_segments` i `work_mem` mogą bardzo istotnie odbiegać od tutaj zalecanych.

Wskazówki dotyczące serwerów współdzielonych

Jeżeli serwer bazy danych współdzieli sprzęt z inną aplikacją — często spotykana jest sytuacja umieszczania wielu aplikacji bazodanowych w pojedynczym systemie — podczas optymalizacji nie można stosować tak agresywnego podejścia, jakie przedstawiono w poprzednim podrozdziale. W takim przypadku dokładna procedura jest trudna do nakreślenia. Należy podjąć próbę przeprowadzenia optymalizacji wartości parametrów związanych z pamięcią i zastosować dolne granice zalecanych wartości:

- parametrowi `shared_buffers` należy przydzielić tylko 10% pamięci RAM, nawet na platformach, na których normalnie zaleca się przydzielenie większej wartości;
- parametrowi `effective_cache_size` należy przydzielić 50% lub mniej pamięci RAM, prawdopodobnie mniej, jeśli wiadomo, że aplikacja będzie używać dużej ilości pamięci;
- trzeba bardzo ostrożnie podchodzić do zwiększania wartości parametru `work_mem`.

Inne sugestie przedstawione w poprzednim podrozdziale mają zastosowanie również tutaj — na przykład użycie większych wartości dla parametru `checkpoint_segments` i odpowiedni wybór wartości parametru `wal_sync_method` nie różnią się niczym w obu systemach.

Następnie trzeba zasymulować działanie aplikacji z pełnym obciążeniem i sprawdzić ilość dostępnej pamięci RAM, aby się przekonać, czy możliwe jest przydzielenie bazie danych większej ilości pamięci. Proces może wymagać kilkakrotnego powtórzenia i na pewno powinien być powiązany z przeprowadzaniem testów wydajności na poziomie aplikacji, o ile jest taka możliwość. W systemie współdzielonym nie ma sensu przydzielanie bazie danych większej ilości pamięci, jeżeli aplikacja bądź inna warstwa buforująca, na przykład na poziomie puli połączeń, nie będzie efektywnie wykorzystywała tej pamięci. Ta sama idea — użycie rozsądnych wartości początkowych, a następnie stopniowe przeprowadzanie optymalizacji na podstawie monitorowania — sprawdza się doskonale także w serwerach dedykowanych.

pgtune

Od PostgreSQL w wersji 8.4 narzędzie pgtune dostępne na stronie *http://pgfoundry.org/projects/pgtune/* może być użyte do utworzenia początkowego pliku konfiguracyjnego *postgresql.conf* dla nowego serwera. Narzędzie pozwala użytkownikowi na zasugerowanie przewidywanego rodzaju obciążenia bazy danych z zakresu od stacji roboczej programisty aż do dedykowanego serwera magazynu danych. Na podstawie tych danych wejściowych oraz parametrów, takich jak ilość pamięci RAM zainstalowana w serwerze, narzędzie generuje plik konfiguracyjny wstępnie zoptymalizowany dla najważniejszych parametrów systemowych, stosuje przy tym metody podobne do omówionych w rozdziale. Uzyskany wynik nie będzie tak dobry jak zastosowanie wskazówek przedstawionych dla serwera dedykowanego i samodzielne przeprowadzanie pomiarów, ale pozwoli na szybkie rozpoczęcie pracy z ogólnie dobrą konfiguracją. Każdy rozsądnie zmodyfikowany plik konfiguracyjny *postgresql.conf* będzie znacznie lepszy od domyślnego, który dostrojony jest jedynie pod kątem alokacji jak najmniejszej ilości pamięci współdzielonej.

Podsumowanie

W konfiguracji bazy danych PostgreSQL istnieje niemal 200 wartości, które można zmodyfikować; poprawne ustawienie w aplikacji wszystkich może być całkiem dużym wyzwaniem. Wskazówki przedstawione w rozdziale powinny pomóc czytelnikowi w rozpoczęciu procesu optymalizacji konfiguracji i uniknięciu najczęściej spotykanych pułapek. Gdy użytkownik napotka problemy, powinny też pokazać, które ustawienia są cenniejsze.

- Wartości domyślne w pliku konfiguracyjnym serwera powodują zapisywanie w dzienniku zdarzeń niewielu informacji i charakteryzują się wyjątkowo skąpymi ustawieniami pamięci. W każdym serwerze należy przynamniej przeprowadzić podstawową optymalizację w celu rozwiązania najgorszych problemów.
- Ustawienia parametrów związanych z pamięcią, czyli przede wszystkim `shared_buffers` i `work_mem`, powinny być przeprowadzane bardzo ostrożnie, aby nie doprowadzić do sytuacji, w której systemowi zabraknie pamięci.
- W celu przygotowania właściwego planu wykonania zapytania planista zapytania musi mieć prawidłowe informacje o pamięci oraz dobre dane statystyczne dotyczące tabeli.
- Demon `autovacuum` ma znaczenie krytyczne dla zapewnienia planiście zapytania odpowiednich informacji. Ponadto demon `autovacuum` zapewnia tabelom uruchamianie właściwych procesów konserwacyjnych.
- W wielu przypadkach wprowadzenie zmiany konfiguracyjnej nie wymaga ponownego uruchomienia serwera, a wiele parametrów można nawet modyfikować dla poszczególnych klientów, zapewniając w ten sposób naprawdę dokładną optymalizację.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Zoptymalizuj swój serwer PostgreSQL

i unikaj problemów, które mogą zmniejszyć jego wydajność!

Mający za sobą już ponad piętnaście lat rozwoju PostgreSQL jest dziś potężnym systemem baz danych typu open source, o sprawdzonej architekturze i reputacji narzędzia niezawodnego oraz nieprzeciętnie wydajnego. Współdziała on ze wszystkimi popularnymi systemami operacyjnymi i jest w pełni zgodny z warunkami ACID. Te zalety sprawiają, że można go używać jako magazynu danych dla aplikacji oraz jako bazy danych dla aplikacji sieciowych. Jednak osiągnięcie maksymalnej wydajności PostgreSQL nie jest wcale zadaniem łatwym, a w trakcie korzystania z jego serwerów można napotkać powtarzające się trudności, zwłaszcza gdy wzrasta obciążenie serwera, a wymagania stają się coraz większe. Jeśli zatem nie chcesz tygodniami dochodzić do właściwych rozwiązań swoich problemów – oto książka, w której znajdziesz całą potrzebną Ci wiedzę.

Masz w rękach kompletny podręcznik, przeznaczony dla średnio i bardzo zaawansowanych administratorów baz danych, którzy już używają PostgreSQL lub dopiero zamierzają to zrobić. Najpierw zapoznasz się z najnowszymi wersjami tej platformy oraz dowiesz się, jak dobierać komponenty serwera, aby optymalnie wykorzystać możliwości systemu. Zobaczysz, jak testować wydajność sprzętu dla bazy danych oraz konfigurować dyski i system plików, aby zwiększać ich efektywność. Poznasz także parametry, których zmiana może powodować problemy, a ponadto najważniejsze ustawienia, ich znaczenie i zasady prawidłowego stosowania. Przeczytasz o tym, jak uzyskać użyteczne wyniki testów wydajności, a także o skutecznym indeksowaniu bazy danych, optymalizacji zapytań i partycjonowaniu danych na podzbiory. Na koniec dowiesz się, jak unikać najczęściej spotykanych problemów i rozwiązywać je, gdy już się pojawią.

Dzięki tej książce:

- poznasz najlepsze praktyki pozwalające na obsłużenie wymagających aplikacji
- odkryjesz, dlaczego sprzęt komputerowy nadaje się (lub nie) dla wysoko wydajnych aplikacji bazodanowych
- zrozumiesz, na czym polegają kompromisy związane z szybkością i niezawodnością działania
- zoptymalizujesz system operacyjny, aby osiągnąć najlepszą wydajność bazy danych
- przeprowadzisz testy wydajności całego systemu, od sprzętu komputerowego po aplikację
- przeanalizujesz rzeczywiste przykłady, co pozwoli Ci poznać wpływ różnych ustawień parametrów serwera na wydajność
- będziesz skutecznie monitorować zdarzenia zachodzące na serwerze, zarówno w bazie danych, jak i poza nią
- znajdziesz najlepsze dodatki, rozszerzające podstawowe możliwości bazy danych PostgreSQL
- dowiesz się, jak przygotować replikację systemów za pomocą najnowszych funkcji wprowadzonych w PostgreSQL 9.0

Nr katalogowy: 5180

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900

0 601 339900

helion.pl księgarnia internetowa

Sprawdź najnowsze promocje:
Książki najchętniej czytane:
Zamów informacje o nowościach:

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena: 79,00 zł

ISBN 978-83-246-3062-2

9 788324 630622

Informatyka w najlepszym wydaniu