Optymalizacja i zaawansowana konfiguracja bazy PostgreSQL

Adam Buraczewski
adamb@nor.pl

Spotkanie WarLUG, 22 stycznia 2004 r.

Plan prezentacji

- Kryteria oceny wydajności
- Konfiguracja systemu Linux i PostgreSQL
- Działanie optymalizatora
- Indeksy
- MVCC i utrzymanie optymalnej wydajności
- Write-Ahead Log
- Prepared Statements i protokół V3
- Nowe algorytmy obsługi cache w 7.5

Cele optymalizacji baz danych

- Zwiększenie liczby realizowanych transakcji na sekundę (TPS)
- Zwiększenie liczby jednocześnie obsługiwanych klientów/aplikacji
- Zmniejszenie obciążenia:
  - serwera
  - sieci i aplikacji klienckiej (jak najmniej danych przesyłanych między komputerami)
- Skrócenie czasu odpowiedzi serwera

Kryteria oceny wydajności

- Benchmarki:
  - dołączone do źródła systemu
  - Benchmarki Open Source
  - Testery specjalizowane, np. Siege
  - Komercyjne oprogramowanie testujące
- Statystyki generowane przez serwer
- Czas odpowiedzi obciążonego serwera, jakość pracy użytkowników
- Własne oprogramowanie testujące

Dostępne benchmarki

- Pgbench:
  - Dołączany standardowo do PostgreSQL (jako contrib)
  - Zasady podobne do TPC-B
  - Test polega na wykonaniu wielu prostych zapytań (INSERT/UPDATE/DELETE) dla tabel zawierających ok. 100 tys. wierszy.
  - Mało reprezentatywny, gdy typowe wykorzystanie bazy to skomplikowane zapytania z dużą liczbą złączeń

Dostępne benchmarki (c.d.)

- Open Source Database Benchmark:
  - http://osdb.sourceforge.net/
  - ok. 40MB danych testowych
  - Wsparcie dla kilku różnych baz danych (w tym PostgreSQL)
  - Zgodny z AS3AP
  - Testy obejmują: wykorzystanie SQL w typowych programach raportujących, aplikachach z GUI itp.

Dostępne benchmarki (c.d.)

- OSDL Database Test Suite:
  - http://www.odsl.net/projects/performance
  - ok. 5MB danych testowych
  - Symulacja:
    - Test 1: Sklep internetowy, serwer WWW pobierający dane z bazy danych (TPC-W)
    - Test 2: System typu magazyn, księgowość, wykorzystanie transakcji (TPC-C)
    - Test 3: Wspomaganie decyzji, hurtownie danych (TPC-H)

Optymalizacja konfiguracji sprzętowej

- Sprzęt lepszej jakości:
  - cache dysków i procesora
  - szybsze dyski, oddzielne podłączenie każdego dysku do kontrolera (lub kilka kontrolerów)
- Zwiększenie ilości: pamięci RAM, procesorów, dysków
- Wykorzystanie macierzy dyskowej (RAID)
  - większa wydajność odczytu
  - większe bezpieczeństwo danych

Konfiguracja Linuksa

- PostgreSQL wykorzystuje:
  - Procesory (każdy klient to oddzielny proces)
  - System plików
  - Cache systemu plików
  - Semafory (ma też własną implementację tzw. Spinlock, dostosowaną do procesora)
  - Pamięć dzieloną (głównie jako własne cache)
  - Pamięć alokowaną dynamicznie

System plików

- Dobór odpowiedniego systemu plików ma ogromne znaczenie
- PostgreSQL ma własne księgowanie danych (WAL), wymaga więc jedynie księgowania metadanych systemu plików
- Najwydajniejszy: ext2 (ext3 podobnie), XFS, reiserfs.
- Przykładowe parametry dla mount:
  `mount -t ext3 -o noatime,data=writeback`

Pamięć dzielona

- Im więcej tym lepiej, ale nie można poświęcić całego RAM (musi działać cache systemu plików) — zwykle ok. 50% RAM.
- Wykorzystywana przez PostgreSQL głównie jako cache
- Od jej ilości zależy wydajność systemu
- Ilość ustawiana poleceniem:
  `sysctl kernel.shmmax=wartość`
- W PostgreSQL <= 7.3 — bardzo mała wartość domyślna

Pamięć dzielona (c.d.)

- Ustawienie ilości pamięci dzielonej używanej przez PostgreSQL (plik postgresql.conf):

```
shared_buffers=10000
```

- W PostgreSQL 7.4 ustawiona domyślnie na 1000 (czyli 8MB) — za mało dla produkcyjnych zastosowań.

Statistic Collector

- Oddzielny proces monitorujący bazy danych
- Pozwala ocenić co się dzieje w serwerze (które tabele są wykorzystywane i w jak często uaktualniane)
- Włączenie poprzez opcję stats_start_collector w postgresql.conf
- Monitoring przez szereg perspektyw zawierających informacje o aktywności serwera

Optymalizator zapytań

- Zwykła ścieżka wykonywania zapytań to: Parser SQL → Planner → Optimizer → Executor
- Działanie: próbowane są wszystkie sposoby wykonania zapytania (np. różna kolejność złączeń), każda jest oceniana i wybierana jest najlepsza
- Polecenie EXPLAIN (EXPLAIN ANALYZE) wyświetlają najlepszą znalezioną ścieżkę (i faktyczny czas wykonania)

Optymalizator zapytań

- Parametry optymalizatora:
  - random_page_cost: wyliczane przez program Bruce'a Momjiana
  - effective_cache_size: wielkość cache w kernelu Linuksa
- Większość parametrów można ustawić programem pg_autotune

Optymalizacja zapytań

- Gdy w złączeniu bierze udział zbyt wiele tabel, optymalizator pracowałby zbyt wolno. Włącza się wówczas optymalizator oparty na algorytmach genetycznych.
- Parametry:
  - geqo_threshold — liczba tabel przy której optymalizator „genetyczny” jest uruchamiany
- Szczegółowy wykład o optymalizacji zapytań — już za kilka tygodni! ;-)

Polecenie ANALYZE

- Zbiera statystyki (histogramy, liczby wartości NULL itp., wielkości tabel) wykorzystywane przez optymalizator do szacowania czasu wykonania zapytań
- Musi być wykonane po każdej większej zmianie tabeli, założeniu lub usunięciu indeksu itp. Dla większej tabeli analizowana jest tylko losowa próbka
- Dokładność sterowana poleceniem ALTER TABLE SET STATISTICS

Znaczenie indeksów

- Przyspieszają wyszukiwanie wierszy spełniających podane kryteria, złączenia
- Ułatwiają sortowanie danych
- Służą do realizacji więzów integralności (PRIMARY KEY, UNIQUE)

Indeksy w PostgreSQL

- 4 typy indeksów:
  - B-Tree — tekst, liczby, inne podstawowe typy danych (99% typowych zastosowań)
  - Hash — podstawowe typy danych
  - R-Tree — typy geometryczne
  - GiST — indeksowanie pełnotekstowe, różne struktury drzewiaste
- Obecnie jedynie indeksy typu B-Tree mają dobrze rozwiązaną pracę wieloużytkownikową (bez blokad)

Indeksy a optymalizacja

- Często należy zakładać indeksy na pola będące kluczami obcymi, co przyspiesza wykonywanie złączeń
- Jeżeli na polu tekstowym wykonywane jest często wyszukiwanie z użyciem wyrażeń regularnych typu: LIKE 'abc%' (znany jest początek wyrażenia), to należy założyć specjalny indeks:

```
create index foo on tabela (pole
text_pattern_ops)
```

Pliki z danymi (tabele, indeksy)

- PostgreSQL nie ma wsparcia dla tablespaces
- Można tworzyć kilka lokalizacji (initlocation) a następnie umieścić bazę danych we wskazanej lokalizacji (np. na innym dysku)
- Tabela zapisana jest zwykle w kilku plikach, podzielonych na bloki 1GB. W oddzielnych plikach są indeksy i TOAST

Pliki z danymi (c.d.)

- Istnieje możliwość przeniesienia pojedynczych plików tabel na inny dysk i zrobienia dowiązania symbolicznego w starym miejscu (tylko przy wyłączonym serwerze PostgreSQL)
- Ryzyko: PostgreSQL może skasować dowiązanie zamiast pliku
- Problemy: pliki o rozmiarze powyżej 1GB, podzielone na mniejsze przez Storage Manager

Multi-Version Concurrency Control

- Jeden z najważniejszych mechanizmów zapewniających bezkonfliktową pracę wielu użytkowników z jedną bazą danych
- Zmniejsza konieczność stosowania blokad
- Ścisła współpraca z mechanizmem transakcji
- Idea działania: może istnieć wiele wersji tego samego wiersza tabeli. Każdy użytkownik „widzi” odpowiednią wersję

MVCC — Działanie

- Każdy wiersz ma ukryte kolumny:
  - Xmin — numer transakcji, w której został stworzony
  - Xmax — numer transakcji, w której został usunięty
  - Cmin — numer polecenia SQL w ramach bieżącej transakcji, w której został stworzony
  - Cmax — numer polecenia SQL w ramach bieżącej transakcji, w której został usunięty
- SELECT xmin, xmax, cmin, cmax FROM foo

MVCC — Działanie

- INSERT — wstawia wiersz z xmax = 0
- DELETE — ustawia xmax na bieżący numer transakcji
- UPDATE = DELETE + INSERT
- Dany użytkownik „widzi” tylko wiersze dla których xmin <= bieżący nr transakcji oraz xmax = 0 lub xmax >= bieżący numer transakcji

MVCC — Ilustracja

- Polecenie INSERT dodaje nowy wiersz:

MVCC — Ilustracja

- Polecenie UPDATE tworzy nową wersję wiersza, ustawia xmin i xmax:

xmin	xmax	Dane
1234	1235	Ala ma kota
1235	0	Ala ma kota

MVCC — Ilustracja

- Polecenie DELETE usuwa wiersz ustawiając jego xmax:

xmin	xmax	Dane
1234	1235	Ala ma kota
1235	1236	Ala ma kota

MVCC — VACUUM

- PostgreSQL pozostawia stare wersje wierszy w tabeli, co powoduje jej rozrastanie
- Konieczność okresowego usuwania wierszy dla których xmax < numer najstarszej transakcji w systemie (polecenie VACUUM)
- Możliwość odzyskania skasowanych wierszy, o ile nie wykonano VACUUM (programem pgfsck)

Rodzaje VACUUM

- Odmiany polecenia VACUUM:
  - VACUUM, tzw. „lazy vacuum” (do zwykłego, codziennego wykonywania) — przesuwa wiersze w obrębie strony, przekazuje puste strony do Free Space Managera
  - VACUUM FULL — przesuwa całe strony, zmniejsza rozmiar pliku, ale blokuje tabelę
  - VACUUM FREEZE — dodatkowo ustawia xmin na 2 (wymagane przy problemach z „przekręceniem się” licznika transakcji w bardzo obciążonych bazach)

Free Space Manager

- Przechowuje informacje o wierszach i stronach odzyskanych przez VACUUM
- Zajmuje część pamięci dzielonej (ale niewiele)
- Obsługuje indeksy typu Btree
- Opcje postgresql.conf:
  - max_fsm_pages
  - max_fsm_relations

FSM — działanie

- Polecenia INSERT/UPDATE pobierają informacje o wolnym miejscu z FSM
- Gdy FSM nie oferuje miejsca, dane dopisywane są na końcu pliku tabeli
- Dane FSM nie są zapamiętywane pomiędzy restartami serwera (warto zrobić VACUUM zaraz po starcie)

FSM — tuning

- Należy założyć pewien procent „martwych” wierszy w tabeli (np. 10–20%)
- Ręczne wykonanie VACUUM VERBOSE, np.:

```
INFO:  Removed 32768 tuples in 8192 pages.
INFO:  Pages 26672: Changed 12288, Empty 0; Tup 32768:
  Vac 32768, Keep 0, UnUsed 41152.
```

- Changed/Pages = 12288/26672 = 46%.
- Należy wykonywać tak często VACUUM żeby ta wartość była jak założona.

FSM — parametry

- max_fsm_relations należy ustawić na tyle ile jest tabel w systemie + zapas
- max_fsm_pages należy ustawić minimum na tyle ile wynosi suma liczb stron „Removed” wyświetlanych przez VACUUM VERBOSE dla wszystkich tabel. Zwyczajowo ustawia się na sumę wartości „Changed” (większą) + zapas

Auto-Vacuum Daemon

- Automatyzuje wykonywanie VACUUM i ANALYZE
- Wymaga włączenia Statistic Collectora dla wierszy
- Uruchamia VACUUM/ANALYZE gdy liczba dodanych/zmienionych/usuniętych wierszy przekroczy podany procent wierszy w tabeli
- Eliminuje potrzebę stosowania crona

Write-Ahead Log

- Realizuje księgowanie (journaling) i współpracuje z mechanizmem transakcji
- Działanie: polecenia INSERT, UPDATE i DELETE dokonują zmian jedynie w plikach WAL.
- Polecenie ROLLBACK: dane transakcji w plikach WAL oznaczone są jako nieważne
- Polecenie COMMIT: dane są oznaczone jako ważne i przeznaczone do przeniesienia do tabel

WAL — działanie

- Pliki WAL (tzw. Segmenty), o wielkości 16MB każdy, znajdują się w katalogu pg_xlog
- Polecenie COMMIT powoduje wywołanie fsync() dla zmienionych plików WAL
- Przeniesienie danych z WAL do tabel odbywa się gdy (konfigurowalne):
  - Zostanie wykonane polecenie CHECKPOINT
  - Zostaną zapełnione min. 3 segmenty
  - Upłynie 5 min.

WAL — działanie (c.d.)

- Po przeniesieniu danych z WAL do tabel wykonywany jest fsync() dla tabel i dane w WAL oznaczone są jako nieważne
- Pliki WAL zawierają sumy kontrolne. Gdy PostgreSQL startuje, odtwarzane są wszystkie zaCOMMITowane transakcje, których sumy kontrolne są poprawne.
- W przypadku zapełnienia dysku transakcje pozostają w plikach WAL. Po starcie systemu, jeżeli jest miejsce, są odtwarzane

WAL — tuning

- Ustawienie parametru fsync na false powoduje zwiększenie wydajności, ale przy awarii (np. zanik zasilania) mniej transakcji zostanie odtworzonych z WAL
- Ustawienie fsync na false nie szkodzi spójności baz danych
- Zmniejszenie częstotliwości checkpointów (zwiększenie checkpoint_timeout oraz checkpoint_segments), zwiększenie liczby plików WAL

WAL — tuning (c.d.)

- Istnieje możliwość opóźnienia fsync() wykonywanego po COMMIT.
- Zwiększenie wydajności poprzez grupowanie wielu COMMITów różnych klientów.
- Parametry:
  - commit_delay — opóźnienie pomiędzy COMMIT a fsync()
  - commit_siblings — minimalna liczba klientów podłączonych do serwera

WAL — tuning (c.d.)

- Można zwiększyć wydajność operacji na plikach WAL poprzez przeniesienie ich na inny dysk niż pliki baz danych
- Przeniesienie:
  - Zatrzymanie postmastera
  - Przeniesienie katalogu pg_xlog
  - Link symboliczny w dotychczasowe miejsce
- Łaty umożliwiające umiejscowienie WAL na partycji bez filesystemu

Prepared Statements

- Umożliwiają jednokrotne przejście etapu: Parser SQL → Optymalizator, a następnie wielokrotne uruchamianie tego samego zapytania z różnymi parametrami, np.:

```
PREPARE foo(integer) AS SELECT * FROM bar
WHERE x = $1; EXECUTE foo(1); EXECUTE foo(2);
```

- Możliwość wywoływania tak przygotowanego zapytania za pomocą libpq (bez użycia polecenia SQL EXECUTE) — dopiero od PostgreSQL 7.4 (protokół komunikacji w wersji 3)

Inne metody zwiększenia wydajności serwera

- Funkcje wykonywane po stronie serwera (SQL, C, PL/SQL, PL/Perl, PL/Python, PL/PHP, PL/Java itp.) — zmniejszenie narzutu związanego z przesyłaniem danych do klienta i z powrotem
- Fast Path — możliwość wywoływania funkcji bez użycia SQL (jedynie libpq)
- Parametr preload_libraries = true — zwiększenie szybkości uruchamiania UDF, kosztem zajętości pamięci

Adaptive Replacement Cache

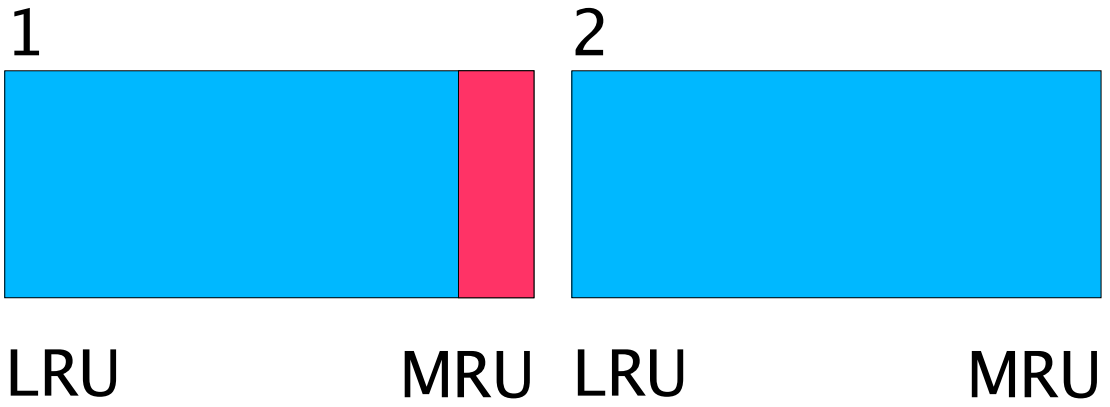
- Prace rozpoczęto w listopadzie 2003 r. (rezultat dopiero w PostgreSQL 7.5)
- Główny koordynator — Jan Wieck
- Problemy z dotychczasowym algorytmem:
  - Polecenie VACUUM niszczyło zawartość cache
  - Sekwencyjny odczyt dużych tabel niszczył zawartość cache
- Problem z poleceniem VACUUM, które obciążało zbytnio system

ARC — działanie

- Algorytm opracowany w IBM Almaden Research Center
- Dwie kategorie stron:
  - jednokrotnie umieszczane w cache (np. strony używane przez odczyty sekwencyjne) — strona startuje jako MRU i powoli przesuwa się do LRU
  - wielokrotnie umieszczane w cache (np. często wykorzystywane tabele) — strona startuje jako LRU i powoli przesuwa się do MRU
  - Strony przechodzą z obszaru 1 do 2 po ponownym użyciu

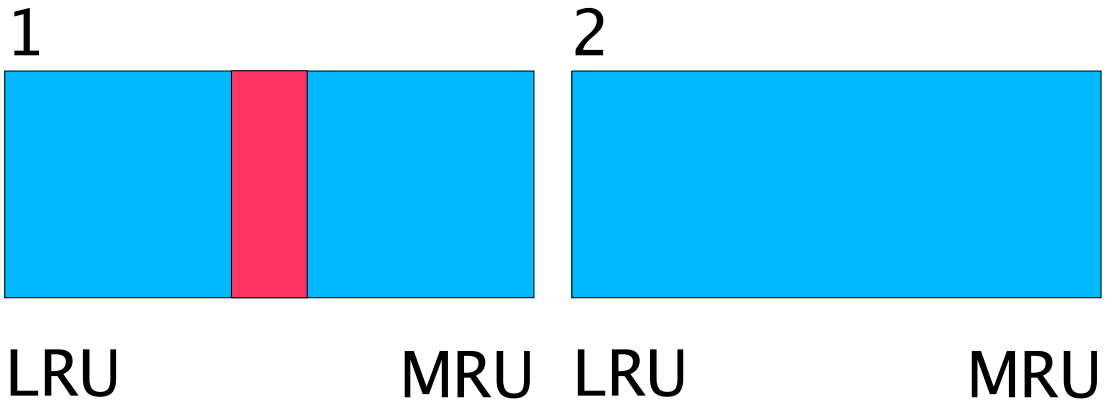
ARC — ilustracja

- Strona, która trafi do cache po raz pierwszy ląduje w obszarze 1 jako MRU:



ARC — ilustracja

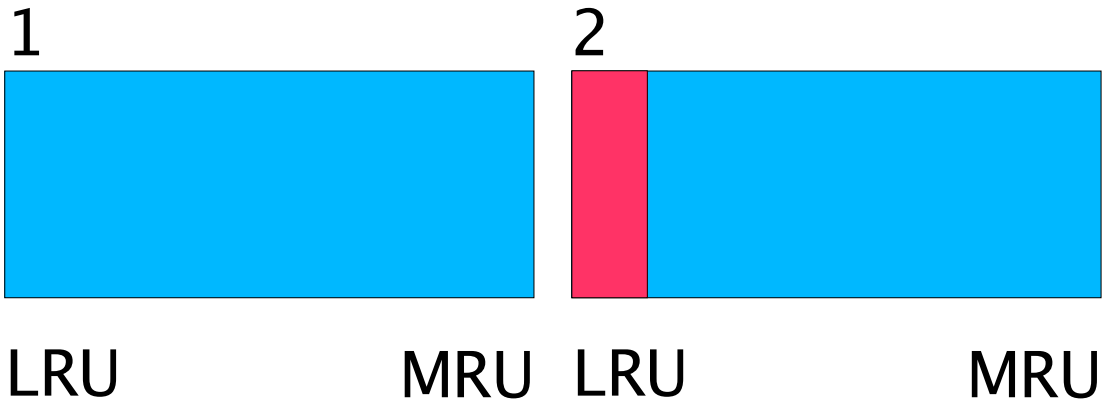
- Jeżeli strona nie jest wykorzystywana, przesuwa się w obszarze 1 do LRU:



- Strony, które dojdą do LRU są usuwane

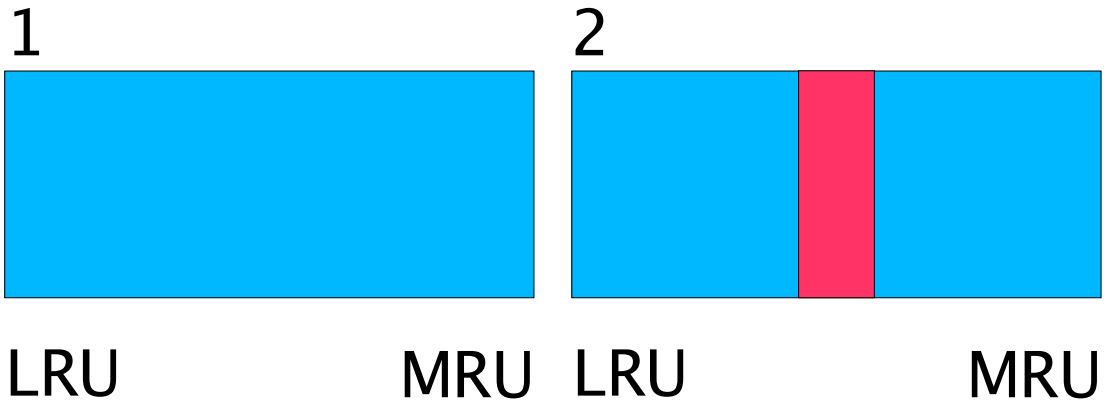
ARC — ilustracja

- Po ponownym użyciu strona trafia do obszaru 2 jako LRU:



ARC — ilustracja

- Kolejne użycia strony powodują jej „wspinanie się” w hierarchii obszaru 2:



ARC — cechy

- Sekwencyjny odczyt tabel oraz VACUUM nie niszczą zawartości głównego cache
- Strony z obszaru 1 są szybko usuwane
- Strony które trafią do obszaru 2 muszą się „piąć w hierarchii” stron najczęściej wykorzystywanch
- Strony rzeczywiście najczęściej wykorzystywane praktycznie nigdy nie są usuwane z cache

ARC — konfiguracja

- Właściwie brak potrzeby konfiguracji samego algorytmu
- Łata rozwiązuje także problem spowolnienia procesu VACUUM („delayed vacuum”) oraz fsync(). Parametry:
  - vacuum_page_delay — spowolnienie VACUUM
  - vacuum_page_groupsize — ilość stron VACUUMowanych za jednym razem
  - lazy_checkpoint_time — spowolnienie wykonywania fsync() dla segmentów WAL

Źródła informacji i linki

- Różne dokumenty nt. PostgreSQL: http://techdocs.postgresql.org/
- Strona Bruce Momjiana: http://momjian.postgresql.org/
- Grupa dyskusyjna pl.comp.bazy-danych
- FAQ: http://pgsql.spsk1.pl
- Archiwum list dyskusyjnych: http://archives.postgresql.org/

Omówione programy pomocznicze

- Pgfsck: http://svana.org/kleptog/pgsql/
- Pgautotune: http://gborg.postgresql.org/
- Pg_autovacuum: contrib/pg_autovacuum
- Randcost: ftp://candle.pha.pa.us/