

# Typy poleceń SQL w Postgresql a Hibernate

Bolesław Ziobrowski

*Artykuł opisuje typy poleceń w JDBC, ich właściwości oraz zastosowanie w narzędziu Hibernate. Poza przedstawieniem wad i zalet ich stosowania, pokazuje główne czynniki decydujące o ich zastosowaniu.*

W interfejsie dostępu do danych języka Java (JDBC) mamy do dyspozycji dwa typy poleceń: Statement oraz PreparedStatement. Według dokumentacji polecenia przygotowane tym różnią się od zwykłych, że przechowują prekompilowane polecenie SQL, które możemy w sposób efektywny wielokrotnie wywołać. Aby wyjaśnić na czym owa prekompilacja polega konieczna jest wiedza o sposobie, w jaki Postgresql przetwarza polecenia.

Proces składa się z następujących etapów:

1. Rozbiór składniowy – w którym najpierw przeprowadzana jest kontrola gramatyczna (czy budowa zapytania jest poprawna), po której następuje kontrola semantyczna (czy tabele, kolumny istnieją, czy użytkownik ma odpowiednie uprawnienia).
2. Przepisywanie – polegające na zmianie treści zapytania: zgodnie z wbudowanymi oraz utworzonymi przez użytkownika regułami (11).
3. Wybór najlepszego planu realizacji polecenia.
4. Wykonanie zapytania według określonego planu

Postgresql nie posiada tak rozbudowanej instrumentacji jak Oracle, więc aby przekonać się, że przetwarzanie poleceń zachodzi według wymienionych kroków, należy wykorzystać mechanizm „dynamicznego śledzenia” (5). Niestety nie jest to tak proste jak uruchamianie „sql trace” i wymaga kompilacji z ustawionymi specjalnymi parametrami samego Postgresql oraz jądra systemu operacyjnego. Po kilku nieudanych próbach z narzędziem SystemTap pod systemem Ubuntu 9.10 z różnymi wersjami jądra, zdecydowałem się na instalację systemu OpenSolaris wewnątrz wirtualnej maszyny VirtualBox. Po początkowych problemach z instalacją systemu zarządzania bazą danych, udało mi się w końcu prześledzić proces przetwarzania polecenia za pomocą narzędzia Dtrace. Pozwala ono między innymi na podłączenie się do konkretnego procesu i automatyczne uruchamianie skryptów napisanych przez nas w specjalnych punktach sondowania wprowadzonych przez programistów, lub jeśli znamy kod źródłowy monitorowanego systemu, prawie w dowolnym jego punkcie. Skrypty te pisane są w specjalnym języku D, podobnym w dużej mierze do języka C. Aby prześledzić interesujący nas proces skorzystamy z istniejących w Postgresql punktów sondowania :

```
query-start(const char *)
query-done(const char *)
query-parse-start(const char *)
query-parse-done(const char *)
query-rewrite-start(const char *)
query-rewrite-done(const char *)
query-plan-start()
query-plan-done()
query-execute-start()
query-execute-done()
```

W języku D możemy definiować akcje, które mogą być wyzwalane po osiągnięciu specjalnych punktów sondowania. Ich składania jest następująca:

```
proces::nazwa_punktu_sondowania
{
    akcja1;
    akcja1;
    ...
}
```

Z informacji zawartych na stronie (9) znamy znaczenie następujących sformułowań języka D:

- `self->nazwa_zmiennej`  
*jest referencją do zmiennej lokalnej wątku*
- `@nazwa[ klucze ] = funkcja-agregująca( argumenty )`  
*pozwała na zapisanie, połączenie i raportowanie danych pochodzących z różnych źródeł. W skrypcie wykorzystywana jest wyłącznie funkcja `count()` zliczająca ile razy osiągnięto dany punkt sondowania.*
- `/predykat/`  
*oznacza warunek, który musi być spełniony aby oznaczona nim akcja została wykonana. W poniższym skrypcie zapewnia on, że akcje wywoływane przy zakończeniu poszczególnych etapów procesu zostaną wywołane, gdy znaczniki czasów rozpoczęcia tych etapów są dostępne (inaczej wyliczanie czasu trwania danego etapu nie ma sensu).*

Poniższy skrypt drukuje w konsoli informacje o zajściu operacji rozbioru składniowego, przepisywania, planowania oraz wykonania poleceń sql wraz z czasem ich trwania.

```
#!/usr/sbin/dtrace -qs
```

```
dtrace::BEGIN
{
    printf("start\n");
}
```

```
dtrace::END
{
    printf("end\n");
}
```

```
postgresql$1:::query-start
{
    @qstart["Query Start"] = count();
    self->queryts = timestamp;
    printf("starting %s\n", copyinstr( arg0 ));
}
```

```
postgresql$1:::query-done
/self->queryts/
{
    printf("query took %d\n", timestamp - self->queryts );
    self->queryts = 0;
}
```

```
postgresql$1:::query-parse-start
{
    @qpstart["parse start"] = count();
```

```

        self->parsets = timestamp;
    }

postgresql$1:::query-parse-done
/self->parsets/
{
    printf("parsing took %d %s\n", timestamp - self->parsets, copyinstr(arg0) );
    self->parsets=0;
}

postgresql$1:::query-rewrite-start
{
    @qpstart["rewrite start"] = count();
    self->rewritets = timestamp;
}

postgresql$1:::query-rewrite-done
/self->rewritets/
{
    printf("rewriting took %d\n", timestamp - self->rewritets );
    self->rewritets=0;
}

postgresql$1:::query-plan-start
{
    @qpstart["Plan start"] = count();
    self->plants = timestamp;
}

postgresql$1:::query-plan-done
/self->plants/
{
    printf("planning took %d\n", timestamp - self->plants );
    self->plants = 0;
}

postgresql$1:::query-execute-start
{
    @qpstart["Execute start"] = count();
    self->executets = timestamp;
}

postgresql$1:::query-execute-done
/self->executets/
{
    printf("executing took %d\n", timestamp - self->executets );
    self->executets = 0;
}

```

Aby prześledzić za pomocą powyższego skryptu sesję psql, wpisujemy w konsoli (w dwóch sesjach):

```
sesja 1) sudo -u postgres postmaster -D ../data &
sesja 1) psql -U postgres -L session.psql
sesja 2) ./tc.d `pgrep -n postgres`
```

Po wykonaniu w psql poniższych poleceń

```
test=# prepare select_index ( varchar ) as select * from pg_indexes where
indexname = $1;
PREPARE
```

```
test=# execute select_index ( 'alamakota' );
 schemaname | tablename | indexname | tablespace | indexdef
-----+-----+-----+-----+-----
(0 rows)
```

```
test=# select * from pg_indexes where indexname = 'alamakota';
 schemaname | tablename | indexname | tablespace | indexdef
-----+-----+-----+-----+-----
(0 rows)
```

obserwujemy pojawienie się w sesji 1 następujących zapisów:

```
starting prepare select_index ( varchar ) as select * from pg_indexes where indexname = $1;
parsing took 655868
rewriting took 53685
planning took 3169665
executing took 8146457
query took 9707928
```

```
starting execute select_index ( 'alamakota' );
parsing took 394
rewriting took 394
executing took 305767
query took 1615882
```

```
starting select * from pg_indexes where indexname = 'alamakota';
parsing took 339016
rewriting took 964337
planning took 4783618
executing took 75740
query took 7041173
```

Widzimy, że zarówno zapytanie zwykle jak i to przygotowane poleceniem PREPARE jest rozbierane składniowo, przepisywane, planowane oraz wykonywane. Mimo że zarejestrowane czasy wykonania są mało miarodajne ( natywny lub wirtualizowany system operacyjny zapewne wielokrotnie przerwał i wznowił badany proces ), to warto zauważyć że podczas przetwarzania polecenia „execute select\_index ( 'alamakota' );” nie występuje etap planowania, zaś etapy rozbioru składniowego i przepisywania polecenia trwają wyjątkowo krótko (na tyle, by nie zostały wstrzymane przez planistę systemów operacyjnych). Wyjaśnienie tego spostrzeżenia znajdziemy w dokumentacji polecenia PREPARE:

*„Polecenie przygotowane jest obiektem bazodanowym, który może zostać wykorzystany do optymalizacji wydajności. Podczas wykonania komendy PREPARE dane polecenie jest rozbierane składniowo, przepisywane i planowane. Przy późniejszym wywołaniu komendy EXECUTE*

*dochodzi wyłącznie do wykonania polecenia. ... Polecenia przygotowane trwają tak długo jak bieżąca sesja. Prekompilowane polecenia są zapomniane z chwilą zakończenia sesji, zatem muszą zostać ponownie utworzone przed ponownym wykorzystaniem. Oznacza to także, że pojedyncze polecenie nie może być wykorzystane przez wielu współbieżnych klientów bazy danych.”*

Gdy już znamy charakterystykę poleceń zwykłych i prekompilowanych, sprawdźmy jak mechanizm ten działa w JDBC. W tym celu wywołamy identyczne polecenie SQL

```
select name, statement, prepare_time from pg_prepared_statements
```

jako typ Statement oraz PreparedStatement i sprawdzimy:

- czy typy te są przetwarzane w inny sposób - na podstawie zapisów w dzienniku zdarzeń sterownika JDBC
- zawartość tabeli pg\_prepared\_statements, która przechowuje wszystkie nazwane polecenia przygotowane przechowywane w bieżącej sesji

Metoda printResultSet, której nie prezentuje by nie rozwlekać przykładu, drukuje wynik zapytania na konsolę w sposób zbliżony do psql.

```
Properties properties = new Properties();
properties.setProperty( "user", "test");
properties.setProperty( "password", "test");
```

```
String sql = "select name, statement, prepare_time from pg_prepared_statements";
```

```
Connection connection = null;
```

```
try
{
```

```
    Driver.setLogLevel( Driver.DEBUG );
    connection = new Driver().connect("jdbc:postgresql://localhost:5432/test",
    properties );
```

```
    System.out.println("normal statement");
    Statement statement = connection.createStatement();
    statement.execute( sql );
    printResultSet(statement);
    statement.close();
```

```
    PreparedStatement preparedStatement = connection.prepareStatement( sql );
    PGStatement pgs = (PGStatement) preparedStatement;
    pgs.setPrepareThreshold( 2 );
```

```
    for ( int i = 1; i <= 3; i++ )
    {
        System.out.println("prepared statement " + i );
        preparedStatement.execute();
        printResultSet(preparedStatement);
    }
```

```
    preparedStatement.close();
}
```

```
finally
```

```
{
    if ( connection != null )
    {
        connection.close();
    }
}
```

Aby zrozumieć poniższy tekst, należy wiedzieć, że sterownik JDBC Postgresql zapisuje w

dzienniku zdarzeń komunikaty, które przesyła do i otrzymuje z serwera. Litery FE są skrótem od Front end – oznaczają wiadomości przesyłane od klienta do serwera, zaś BE (Back end) oznaczają wiadomości przesyłane od serwera do klienta.

Po uruchomieniu powyższego kodu otrzymujemy (dla zwiększenia czytelności usunąłem zbędne wpisy):

normal statement

```
FE=> Parse(stmt=null,query="select name, statement, prepare_time
      from pg_prepared_statements",oids={})
FE=> Bind(stmt=null,portal=null)
FE=> Describe(portal=null)
FE=> Execute(portal=null,limit=0)
FE=> Sync
<=BE ParseComplete [null]
<=BE BindComplete [null]
<=BE RowDescription(3)
<=BE CommandStatus(SELECT)
<=BE ReadyForQuery(I)
```

prepared statement 1

```
FE=> Parse(stmt=null,query="select name, statement, prepare_time
      from pg_prepared_statements",oids={})
FE=> Bind(stmt=null,portal=null)
FE=> Describe(portal=null)
FE=> Execute(portal=null,limit=0)
FE=> Sync
<=BE ParseComplete [null]
<=BE BindComplete [null]
<=BE RowDescription(3)
<=BE CommandStatus(SELECT)
<=BE ReadyForQuery(I)
```

prepared statement 2

```
FE=> Parse(stmt=S_1,query="select name, statement, prepare_time
      from pg_prepared_statements",oids={})
FE=> Bind(stmt=S_1,portal=null)
FE=> Describe(portal=null)
FE=> Execute(portal=null,limit=0)
FE=> Sync
<=BE ParseComplete [S_1]
<=BE BindComplete [null]
<=BE RowDescription(3)
<=BE DataRow
<=BE CommandStatus(SELECT)
<=BE ReadyForQuery(I)
```

```
name | statement | prepare_time |
S_1 | select name, statement, prepare_time from
pg_prepared_statements | 2009-11-30 13:57:25.074468 |
```

prepared statement 3

```
FE=> Bind(stmt=S_1,portal=null)
```

```
FE=> Describe(portal=null)
FE=> Execute(portal=null,limit=0)
FE=> Sync
<=BE BindComplete [null]
<=BE RowDescription(3)
<=BE DataRow
<=BE CommandStatus(SELECT)
<=BE ReadyForQuery(I)
```

```
name | statement | prepare_time |
S_1 | select name, statement, prepare_time from
pg_prepared_statements | 2009-11-30 13:57:25.074468 |
```

Porównując opisy przetwarzania pierwszych dwóch poleceń widzimy, że są wykonywane w identyczny sposób: najpierw przesyłane jest żądanie przeprowadzenia wstępnego przetworzenia polecenia – **Parse**, potem przesyłane są ewentualne parametry w komunikacie **Bind** (stąd właśnie bierze się odporność zapytań przygotowanych na ataki wstrzykiwania SQL (SQL Injection) – argumenty wywołania nie mogą zostać dzięki temu wzięte za fragment polecenia), następnie **Describe**, które prosi serwer o przesłanie metadanych opisujących zwracany wynik, dalej – komenda **Execute**, nakazująca wykonanie polecenia. Komunikat **Sync** oznacza koniec przetwarzania polecenia przez klienta i umożliwia synchronizację klienta z serwerem w razie wystąpienia jakiegoś błędu. Komunikaty przesyłane przez serwer to: **ParseComplete** oraz **BindComplete** oznaczające pomyślne wykonanie pierwszych dwóch poleceń klienta, następnie **RowDescription** który zawiera opis zwracanego wyniku, zwracany jako odpowiedź na polecenie **Describe**, dalej status polecenia – **CommandStatus**, oraz ogłoszenie gotowości do przyjęcia kolejnego polecenia – **ReadyForQuery**. Protokół komunikacyjny w oparciu o który przetwarzane są oba zapytania, zwany *protokołem rozbudowanych (rozszerzonych) zapytań* (Extended Query Protocol) opisany jest w rozdziale 45.2.3 dokumentacji PostgreSQL. Istnieje również protokół dla prostych zapytań, którego jednakże nie będę tutaj opisywał, gdyż jak widać sterownik JDBC korzysta wyłącznie z tego pierwszego.

Kolejną ważną a przy tym nieoczekiwaną kwestią jest pusty wynik zapytania przygotowanego. Oznacza to, że pierwsze wywołanie nie spowodowało utworzenia zapytania przygotowanego na serwerze. W dalszej części raportu zdarzeń widzimy, że dopiero w drugiej iteracji pętli – podczas drugiego wykonania obiektu PreparedStatement zostaje utworzony odpowiedni obiekt po stronie serwera, widniejący w tabeli `pg_prepared_statements` pod nazwą `S_1`. Dalej widzimy, że począwszy od trzeciego wywołania, nie jest już przesyłany komunikat **Parse**. Owo zaskakujące zachowanie sterownika wynika oczywiście z wywołania metody `PGStatement.setPrepareThreshold()`, która wyznacza próg wywołań obiektu klasy PreparedStatement, przy którym dochodzi do utworzenia nazwanego polecenia prekompilowanego po stronie serwera. Co więcej, mechanizm ten jest domyślnie włączony, przy czym próg jest osiągany dopiero przy – niespodzianka - piątym wywołaniu. Jawne nadanie wartości progowi służy wyłącznie skróceniu dziennika zdarzeń. Co ważne, mechanizm ten możemy konfigurować również dla konkretnego połączenia metodą `PGConnection.setPrepareThreshold()` lub źródła danych za pomocą parametru `prepareThreshold` (7). Skoro posiadamy już podstawową wiedzę na temat działania sterownika dostępu do danych, przejdź o poziom wyżej, do narzędzia Hibernate i sprawdźmy jako ono się spisuje w badanym obszarze.

Test jaki przeprowadzimy na narzędziu mapowania obiektowo – relacyjnego polega na zapisie kilku obiektów klasy Entry, a następnie na kilkukrotnym odpytaniu bazy, przy włączonym tworzenia serwerowych poleceń prekompilowanych już przy pierwszym wykonaniu. Operację przeprowadzimy na „gołym” Hibernate, bez żadnych dodatków – czyli tak jak w poprzednich artykułach. Klasa Entry jest następującej treści:

```

@Entity
public class Entry
{
    @Id
    @GeneratedValue
    private Integer id;

    private String content;

    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    private String email;

    Entry() { /* required by hiberante */ }

    public Entry( String content, String email )
    {
        this.content = content;
        creationDate = new Date();
        this.email = email;
    }

    public Integer getId()
    {
        return id;
    }
}

```

Dalej – równie nieskomplikowany kod testu:

```

Session session = HibernateUtility.getSession( persistentClasses );
Transaction transaction = session.beginTransaction();
transaction.begin();

session.doWork( new Work()
{
    @Override
    public void execute(Connection connection) throws SQLException
    {
        PGConnection pgc = (PGConnection)connection;
        pgc.setPrepareThreshold(1);
    }
});

session.persist( new Entry( "Stefan", "stefan@tralala.com" ) );
session.persist( new Entry( "Ziutek", "ziutek@tralala.com" ) );

session.flush();

session.createQuery("from Entry e where e.email like '%@tralalala.com'").list();
printPreparedStatements(session);

session.createQuery("from Entry e where e.email like '%@tralalala.com'").list();
printPreparedStatements(session);

transaction.rollback();
session.close();

```

Metoda printPreparedStatement:

```

private static void printPreparedStatements(Session session)
{
    List<Object[]> statements = session.createSQLQuery("select " +
        „name,statement,prepare_time „ +
        „from pg_prepared_statements order by prepare_time asc").list();
    for ( Object[] statement : statements )
    {
        System.out.println( Arrays.toString( statement ) );
    }
}

```

Po uruchomieniu testu ( przy włączonych ustawieniach SHOW\_SQL i FORMAT\_SQL ) otrzymujemy:

```

Hibernate:
select
    entry0_.id as id0_,
    entry0_.content as content0_,
    entry0_.creationDate as creation3_0_,
    entry0_.email as email0_,
    entry0_.isDeleted as isDeleted0_
from
    Entry entry0_
where
    entry0_.email like '%@tralalala.com'

```

```

Hibernate:
select
    name,
    statement,
    prepare_time
from
    pg_prepared_statements
order by
    prepare_time asc

```

```

[S_1, BEGIN, 2009-12-01 15:10:31.792254]
[S_6, select name,statement,prepare_time from pg_prepared_statements order by
prepare_time asc, 2009-12-01 15:10:32.027655]

```

```

Hibernate:
select
    entry0_.id as id0_,
    entry0_.content as content0_,
    entry0_.creationDate as creation3_0_,
    entry0_.email as email0_,
    entry0_.isDeleted as isDeleted0_
from
    Entry entry0_
where
    entry0_.email like '%@tralalala.com'

```

```

Hibernate:
select
    name,
    statement,
    prepare_time
from
    pg_prepared_statements
order by
    prepare_time asc

```

```

[S_1, BEGIN, 2009-12-01 15:10:31.792254]
[S_8, select name,statement,prepare_time from pg_prepared_statements order by
prepare_time asc, 2009-12-01 15:10:32.037425]

```

Jak widzimy po zawartości tabeli `pg_prepared_statements` – JDBC posłusznie tworzy za każdym razem prawdziwe nazwane polecenie przygotowane, ale nic to nam nie daje, gdyż Hibernate przy każdym zapytaniu otwiera i zamyka nowy obiekt `PreparedStatement`, co niweczy efekt wywołania `pgc.setPrepareThreshold(1)`. Bierze się to stąd, że narzędzie Hibernate samo w sobie nie buforuje poleceń. Aby rozwiązać ten problem musimy skorzystać z puli połączeń – takiej jak na przykład C3P0. Dlatego też dołączamy bibliotekę w wersji 0.9.1.2 i odpowiednio konfigurujemy narzędzie ORM (wbrew pozorom nie jest takie proste), ustawiając poniższe parametry puli:

```
props.put( "maxPoolSize", 5 );
props.put( "minPoolSize", 0 );
props.put( "maxStatementsPerConnection" , 100 );
props.put( "maxStatements" , 100 );
props.put( "initialPoolSize" , 0 );
props.put( "maxIdleTime", 120 );
props.put( "idleConnectionTestPeriod", 100 );
props.put( "acquireIncrement", 1 );
```

Pogrubioną czcionką oznaczyłem parametry o największym wpływie na mechanizm poleceń przygotowanych. Aby uruchomić przykład musimy jeszcze zamienić fragment :

```
public void execute(Connection connection) throws SQLException
{
    PGConnection pgc = (PGConnection)getField(
        ((NewProxyConnection)connection), "inner");
    pgc.setPrepareThreshold(1);
}
```

gdyż pula C3P0 opakowuje połączenia swoimi klasami i nie udostępnia sposobu jego uzyskania inaczej niż przez mechanizm releksji języka Java, co robi metoda `getField`. Przy ponownym uruchomieniu przykładu ostatnie odpytanie tabeli `pg_prepared_statement` zwraca nam:

```
[S_1, BEGIN, 2009-12-01 15:48:22.620031]
[S_2, select nextval ('hibernate_sequence'), 2009-12-01 15:48:22.620207]
[S_3, insert into Entry (content, creationDate, email, isDeleted, id) values
($1, $2, $3, $4, $5), 2009-12-01 15:48:22.658155]
```

```
[S_4, select entry0_.id as id0_, entry0_.content as content0_,
entry0_.creationDate as creation3_0_, entry0_.email as email0_,
entry0_.isDeleted as isDeleted0_ from Entry entry0_ where entry0_.email like
'%@tralalala.com', 2009-12-01 15:48:22.912218]
```

```
[S_5, select name,statement,prepare_time from pg_prepared_statements order by
prepare_time asc, 2009-12-01 15:48:22.933192]
```

Co oznacza, że pula połączeń buforuje polecenia zgodnie z zaprezentowanymi wcześniej ustawieniami. Wniosek płynie z tego taki, że aby skorzystać z zalet poleceń prekompilowanych należy skorzystać z puli połączeń i jawnie ustawić rozmiar bufora poleceń: globalnego dla źródła danych i/lub lokalnego dla pojedynczego połączenia. W wielu pulach mechanizmy te są domyślnie wyłączone.

No dobrze – wiemy już, jak działają polecenia przygotowane i co należy zrobić, by skorzystać z ich dobrodziejstwa. Zanim jednak zaczniemy konfigurować bufory poleceń w pulach połączeń naszych aplikacji należy zadać sobie pytanie – skoro mechanizm ten jest taki świetny to czemu autorzy sterowników JDBC dla Postgresql zdecydowali się na to by domyślnie prawdziwa kompilacja następowała dopiero przy piątym wywołaniu zapytania, a wiele pul domyślnie wyłącza buforowanie obiektów `PreparedStatement` ? W tym momencie zaczynamy podejrzewać, że polecenia te muszą posiadać jakiś feler. I jest to dobry trop. Aby wyjaśnić na czym on polega

musimy wrócić do opisu komendy PREPARE w dokumentacji wykorzystywanej bazy danych. Skoro rozbiór składniowy, przepisywanie i optymalizacja planu wykonania następują tylko raz, to nawet gdy dane polecenie wykonamy dla zupełnie innych parametrów, plan się nie zmieni. Co więcej, podczas poszukiwania najlepszego planu optymalizator zapytań nie bierze pod uwagę parametrów wywołania polecenia. Jak wielkie znaczenie ma ten fakt dla wydajności naszych aplikacji przekonać się możemy na kilku przykładach.

W pierwszym założymy, że posiadamy prostą tabelę zawierającą 10000 wierszy - listów, w tym około 3% nie przeczytanych.

```
create table email
(
    email_id int primary key,
    email_from varchar(100),
    email_content char(1000),
    email_status varchar(20),
    email_created timestamp default current_timestamp,
    email_modified timestamp default current_timestamp
);

create sequence email_id_sequence;

insert into email( email_id, email_from, email_content, email_status )
select      nextval('email_id_sequence'),
           floor((random() * 10000)) || '@test.pl',
           'alamakota',
           case when random() > 0.03 then 'read' else 'unread' end
from generate_series(1,10000) as s(a);
INSERT 0 10000

analyze email;
```

Jako, że wiersze te zawierają pole char(1000), a blok bazy danych ma domyślny rozmiar 8kB, wiemy że tabela będzie miała wielkość kilku tysięcy bloków. Dokładne dane otrzymujemy wykonując zapytanie:

```
SELECT relname, reltuples, relpages, reltoast
FROM pg_class
where relname = 'email';
```

```
relname | reltuples | relpages
-----+-----+-----
email   |      10000 |      1429
```

A teraz zadanie domowe dla ciekawskich : ile stron będzie posiadać nasza tabela, gdy w definicji zmienimy definicję email\_content na char(3000) ? Czy będzie większa, mniejsza, czy też nie ulegnie żadnym zmianom ? I co najważniejsze – dlaczego ?

Założymy teraz, że w pewnym fragmencie naszej aplikacji potrzebujemy odpytać owe nieprzeczytane listy. Aby przyspieszyć zapytanie konstruujemy indeks częściowy (partial index), który zawiera wpisy wyłącznie dla listów nieprzeczytanych. Tworzymy go w następujący sposób:

```
create index unread_email_index on email ( email_id ) where email_status = 'unread';
```

W końcu odpytujemy naszą tabelę o listy przeczytane:

```
explain select * from email where email_status = 'read';
```

#### QUERY PLAN

```
-----  
Seq Scan on email (cost=0.00..1554.00 rows=9698 width=1041)  
  Filter: ((email_status)::text = 'read'::text)  
(2 rows)
```

oraz nieprzeczytane:

```
explain select * from email where email_status = 'unread';
```

#### QUERY PLAN

```
-----  
Index Scan using unread_email_index on email (cost=0.00..60.54 rows=302  
width=1041)  
(1 row)
```

Jak widać pierwsze z zapytań powoduje pełne skanowanie tabeli, a drugie – indeksu, zatem są wykonywane w optymalny sposób. Teraz sprawdzimy, jaki plan będzie miało zapytanie, w którym stan listu jest parametrem:

```
prepare select_email( varchar ) as select * from email where email_status = $1;
```

```
explain execute select_email( 'read' );
```

#### QUERY PLAN

```
-----  
Seq Scan on email (cost=0.00..278.00 rows=5000 width=12041)  
  Filter: ((email_status)::text = ($1)::text)  
(2 rows)
```

```
explain execute select_email( 'unread' );
```

#### QUERY PLAN

```
-----  
Seq Scan on email (cost=0.00..278.00 rows=5000 width=12041)  
  Filter: ((email_status)::text = ($1)::text)  
(2 rows)
```

Widzimy, że w obu przypadkach stosowany jest ten sam plan wykonania, który jest wydajny wyłącznie dla pierwszego z nich, a użyty w drugim będzie wielokrotnie wolniejszy od dostępu przez indeks `unread_email_index`. Jak wspomniałem wcześniej, dzieje się tak dlatego, że optymalizator nie bierze pod uwagę podanych mu parametrów, więc nie jest w stanie skorzystać z histogramów określających częstości występowania wartości w kolumnie `email_status`. Z tego powodu optymalizator zakłada, że rozkład wartości jest równomierny i spodziewa się znaleźć 5000 rekordów spełniających predykat `email_status = $1` – pokazuje to atrybut **rows**.

Wszystko to oznacza, że indeks częściowy `email_status_index` jest w tej sytuacji bezużyteczny.

Poprzedni przykład pokazywał, jak wskutek stosowania poleceń przygotowanych popsuć można ścieżkę dostępu do jednej tabeli. Teraz zobaczymy, że w podobny sposób doprowadzić można do wyboru złego sposobu złączenia tabel. W tym celu utworzymy nową tabelę przechowującą załączniki do listów.

```
create table email_attachment  
(  
  attachment_id int primary key,  
  attachment_email_id int references email( email_id ),  
  attachment_content bytea  
);
```

```
create index attachment_email_id_fkey on email_attachment ( attachment_email_id );
```

```
create sequence attachment_id_sequence;
```

Następnie wypełniamy ją danymi – tworzymy po dwa załączniki dla każdego listu.

```
insert into email_attachment ( attachment_id, attachment_email_id )
select nextval('attachment_id_sequence'), email_id
from email, generate_series(1,2);
INSERT 0 20000
```

Usuwamy niepotrzebny indeks,

```
drop INDEX unread_email_index ;
```

i dodajemy nowe do tabeli email oznaczające czy list został usunięty. Wykonujemy to po tym, by utworzyć kolumnę o jeszcze bardziej „wykrzywionym” rozkładzie wartości niż email\_status.

```
alter table email add column email_is_deleted boolean default 'false';
```

```
update email set email_is_deleted = TRUE
where email_id in ( select floor((random()*10000)) from generate_series(1,10) );
UPDATE 10
```

```
analyze verbose email;
analyze email_attachment;
```

Widzimy, że tylko 10 rekordów w tabeli email jest oznaczonych jako usunięte. Następnie tworzymy indeks pozwalający na szybki dostęp do owych skasowanych listów :

```
create index email_is_deleted_index on email( email_id, email_is_deleted );
```

i zadajemy proste pytanie łączące obie tabele:

```
explain select *
from email
join email_attachment on email_id = attachment_email_id
where email_is_deleted = TRUE;
```

#### QUERY PLAN

```
-----
Nested Loop (cost=0.00..322.27 rows=20 width=1082)
-> Index Scan using email_is_deleted_index on email (cost=0.00..215.14 rows=10
width=1042)
    Index Cond: (email_is_deleted = true)
    Filter: email_is_deleted
-> Index Scan using attachment_email_id_fkey on email_attachment (cost=0.00..10.69
rows=2 width=40)
    Index Cond: (email_attachment.attachment_email_id = email.email_id)
(6 rows)
```

W wybranym przez optymalizator planie najpierw skanowany jest cały indeks email\_is\_deleted, z którego pobierane są wpisy dla „usuniętych” listów. Po napotkaniu takiego wpisu wyszukane zostają, poprzez skan na indeksie attachment\_email\_id\_fkey wszystkie załączniki, które do niego należą. Tak w ogólnym zarysie przebiega złączenie tabel email i email\_attachment według algorytmu pętli zagnieżdżonej (Nested Loop). Jest to plan optymalny. Aby się o tym przekonać – wystarczy użyć komendy explain analyze, która oprócz wygenerowania planu zapytania także je wykonuje i podaje rzeczywisty czas wykonania, liczbę rekordów oraz przeciętny ich rozmiar dla każdego etapu przetwarzania. Czas wykonania wyniósł w tym przypadku 2 milisekundy.

Następnie zadamy pytanie o pozostałe (nieusunięte) listy :

```
explain select * from email join email_attachment on email_id =
attachment_email_id where email_is_deleted = FALSE;
```

#### QUERY PLAN

```
-----
Merge Join (cost=0.00..1546.85 rows=19980 width=1082)
  Merge Cond: (email.email_id = email_attachment.attachment_email_id)
    -> Index Scan using email_pkey on email (cost=0.00..409.84 rows=9990 width=1042)
        Filter: (NOT email_is_deleted)
    -> Index Scan using attachment_email_id_fkey on email_attachment (cost=0.00..862.23
        rows=20000 width=40)
(5 rows)
```

Teraz dla odmiany optymalizator zdecydował się na złączenie obu tabel algorytmem scalającym (Merge Join). Co ciekawe zarówno listy jak i załączniki pobierane są przez indeksy na kluczach głównych tabeli, a nie – jak można by się było spodziewać – za pomocą pełnego skanowania. Po wyłączeniu dostępu przez skanowanie indeksu poleceniem :

```
set enable_indexscan= false;
```

plan jest następujący:

#### QUERY PLAN

```
-----
Hash Join (cost=2956.88..5137.68 rows=19980 width=1082)
  Hash Cond: (email_attachment.attachment_email_id = email_part.email_id)
    -> Seq Scan on email_attachment (cost=0.00..289.00 rows=20000 width=40)
    -> Hash (cost=1529.00..1529.00 rows=9990 width=1042)
        -> Seq Scan on email_part (cost=0.00..1529.00 rows=9990 width=1042)
            Filter: (NOT email_is_deleted)
```

Mimo że wygląda na bardziej obiecujący to w rzeczywistości jego wykonanie zajmuje prawie o połowę więcej czasu (370 ms) od planu oryginalnego (280 ms). Prawdopodobnie wynika to z tego, że przy takiej ilości danych bardziej opłacalne jest uniknięcie sortowania i złączenie przez scalanie, niż potencjalnie szybszy sekwencyjny odczyt danych całych tabel. Po dokładniejszym sprawdzeniu okazało winowajcą jest tutaj zbyt niskie ustawienie pamięci dostępnej przeznaczonej na sortowanie i mieszanie (hashing) rekordów. Domyślnie ma ono wartość 1 MB. Gdy obszar jest zbyt mały to dochodzi do sortowania/mieszania wielofazowego z udziałem dysku twardego, wielokrotnie wolniejszego od operacji przeprowadzonej w całości w pamięci. Po wykonaniu polecenia

```
set work_mem = 2000;
```

zwiększającego omawiany obszar do wielkości 2 MB, optymalizator decyduje się na złączenie mieszające, przy czym teraz czas realizacji zapytania wynosi teraz 240 ms. Poniższe zapytania przeprowadzone są jednak przy domyślnej wartości ustawienia work\_mem.

Wracając do tematu – przygotowujemy następujące zapytanie :

```
prepare select_email( boolean )
as select *
from email
join email_attachment on email_id = attachment_email_id
where email_is_deleted = $1;
```

I sprawdzamy plany wykonania dla listów „usuniętych”:

```
explain execute select_email( TRUE );
```

#### QUERY PLAN

```
-----  
Merge Join (cost=0.00..1459.58 rows=10000 width=1082)  
Merge Cond: (email.email_id = email_attachment.attachment_email_id)  
-> Index Scan using email_pkey on email (cost=0.00..434.84 rows=5000 width=1042)  
    Filter: (email_is_deleted = $1)  
-> Index Scan using attachment_email_id_fkey on email_attachment (cost=0.00..862.23  
    rows=20000 width=40)  
(5 rows)
```

oraz „istniejących” :

```
explain execute select_email( FALSE );
```

#### QUERY PLAN

```
-----  
Merge Join (cost=0.00..1459.58 rows=10000 width=1082)  
Merge Cond: (email.email_id = email_attachment.attachment_email_id)  
-> Index Scan using email_pkey on email (cost=0.00..434.84 rows=5000 width=1042)  
    Filter: (email_is_deleted = $1)  
-> Index Scan using attachment_email_id_fkey on email_attachment  
    (cost=0.00..862.23 rows=20000 width=40)  
(5 rows)
```

Podobnie jak poprzednim przykładzie widzimy, że optymalizator wybrał plan bez uwzględnienia wartości argumentów i założył, że tabela zawiera po 5000 rekordów „usuniętych” i „istniejących”. Pytanie o „usunięte” rekordy trwa teraz dużo dłużej, bo nie 2, lecz 120 milisekund (według EXPLAIN ANALYZE).

Kolejny, ostatni, przykład pokazuje wpływ poleceń przygotowanych na wydajność zapytań na tabelach partycjonowanych. W PostgreSQL partycjonowanie tabel działa w oparciu o 'obiektywny' mechanizm dziedziczenia. Polega to na tym, że najpierw definiujemy tabelę bazową, a następnie dla każdej partycji tabelę pochodną - z użyciem specjalnej klauzuli `inherits`. Dokładny opis partycjonowania tabel znaleźć można w (15). W bieżącym przykładzie wygląda to następująco :  
deklarujemy tabelę bazową w zwyczajny sposób

```
create table email  
(  
    email_id int primary key,  
    email_from varchar(100),  
    email_content char(1000),  
    email_status varchar(20),  
    email_created timestamp default current_timestamp,  
    email_modified timestamp default current_timestamp  
);
```

a następnie tworzymy trzy partycje, przechowujące listy z roku 2008, 2009 oraz 2010. Co ważne, każda tabela - partycja musi definiować ograniczenie pozwalające kompilatorowi określić jakiego 'rodzaju' dane ona przechowuje.

```
create table email_2008  
(  
    check ( email_created >= DATE '2008-01-01' and email_created < DATE '2009-01-01' )  
) inherits (email);  
  
create table email_2009  
(  
    check ( email_created >= DATE '2009-01-01' and email_created < DATE '2010-01-01' )  
) inherits (email);  
  
create table email_2010  
(  
    check ( email_created >= DATE '2010-01-01' and email_created < DATE '2011-01-01' )  
) inherits (email);
```

```
analyze email;
analyze email_2008;
analyze email_2009;
analyze email_2010;
```

Bez wstawiania jakichkolwiek danych zadajemy następnie pytanie:

```
explain select *
  from email
  where email_created > DATE '2009-01-01'
        and email_created < DATE '2009-02-03';
```

#### QUERY PLAN

```
-----
Result (cost=0.00..20.00 rows=2 width=12300)
-> Append (cost=0.00..20.00 rows=2 width=12300)
   -> Seq Scan on email (cost=0.00..10.00 rows=1 width=12300)
       Filter: ((email_created > '2009-01-01'::date) AND (email_created < '2009-02-03'::date))
   -> Seq Scan on email_2009 email (cost=0.00..10.00 rows=1 width=12300)
       Filter: ((email_created > '2009-01-01'::date) AND (email_created < '2009-02-03'::date))
(6 rows)
```

i widzimy, że optymalizator skanuje tylko pustą tabelę bazową email oraz partycję email\_2009 która jako jedyna przechowuje rekordy z roku 2009-ego. Optymalizator wybiera te partycje a pomija pozostałe właśnie w oparciu o nałożone na tabelę ograniczenia zakresów dat. Operacja ta, nazywana przycinaniem partycji (partition pruning), może znacząco przyspieszyć przetwarzanie zapytań na tabelach partycjonowanych. Teraz zobaczmy co się stanie, gdy zapytanie te prekompilujemy :

```
prepare select_email ( timestamp, timestamp ) as select * from email where
email_created > $1 and email_created < $1;
```

```
explain execute select_email( DATE '2009-01-01', DATE '2009-02-03' );
```

#### QUERY PLAN

```
-----
Result (cost=0.00..40.00 rows=4 width=1300)
-> Append (cost=0.00..40.00 rows=4 width=1300)
   -> Seq Scan on email (cost=0.00..10.00 rows=1 width=1300)
       Filter: ((email_created > $1) AND (email_created < $1))
   -> Seq Scan on email_2008 email (cost=0.00..10.00 rows=1 width=1300)
       Filter: ((email_created > $1) AND (email_created < $1))
   -> Seq Scan on email_2009 email (cost=0.00..10.00 rows=1 width=1300)
       Filter: ((email_created > $1) AND (email_created < $1))
   -> Seq Scan on email_2010 email (cost=0.00..10.00 rows=1 width=1300)
       Filter: ((email_created > $1) AND (email_created < $1))
(10 rows)
```

Zamiast jednej partycji skanowane są wszystkie partycje. Nie jest to wszakże dziwne biorąc pod uwagę, że optymalizator nic nie wie o argumentach wywołania i tym samym nie ma żadnej podstawy do pominięcia którejkolwiek z partycji.

Wiemy już, że użycie polecenia prekompilowanego doprowadzić może do wyboru niewydajnego sposobu: dostępu do tabeli, złączenia tabel oraz skanowania tabeli partycjonowanej. Warto w tym miejscu zadać sobie pytanie, czemu opisywany system zarządzania bazą danych zachowuje się w taki sposób i ignoruje wartości argumentów wywołań poleceń prekompilowanych. Czy nie mógłby zachować się bardziej „przyjaźnie” i wziąć ich pod uwagę ? Podejście to zostało zaimplementowane w systemie Oracle i znane jest pod nazwą *podglądania wartości zmiennych wiązanych* (bind variables peeking). Możliwe, że spowodowało by to obranie planów optymalnych dla pierwszego wywołania, ale co z następnymi ? Gdyby zastosować to podejście w powyższych

przykładach to wcale nie znaleźlibyśmy się lepszej sytuacji – a całkiem możliwe, że w gorszej bo :

1. dostęp do znacznej większości rekordów w tabeli z użyciem indeksu jest dużo mniej efektywne od sekwencyjnego pełnego skanowania
2. algorytm złączenia z pętlą zagnieżdżoną zadziała bardzo wolno dla dużej liczby rekordów w pętli zewnętrznej
3. jeśli przy pierwszym wywołaniu przytniemy partycje wymagane w kolejnych wywołaniach to baza danych zwróci niepoprawny wynik

W Oracle (9i i 10g) problem ten o tyle poważny, że plany poleceń przechowywane są globalnie dostępnym fragmencie puli współdzielonej (shared pool) i, jak sama nazwa wskazuje, mogą być wykorzystywane przez wiele współbieżnych sesji. W sytuacji gdy użytkownik A wykona polecenie przygotowane to w puli współdzielonej zostanie umieszczony plan zoptymalizowany pod kątem podanych przez niego parametrów. Teraz gdy użytkownik B wywoła identyczne tekstowo (identyczne polecenie w języku SQL – łącznie z nazwami zmiennych wiązanych) to wykorzystany zostanie istniejący plan zoptymalizowany dla użytkownika A. Jeśli natomiast najpierw zapytanie wykona użytkownik B, to A dostanie plan potencjalnie dla niego nieoptymalny. Problem ten rozwiązano w wersji 11g (16), ale zapewne wiele osób ma jeszcze do czynienia z poprzednimi wersjami, dlatego podaję łącznie do dokładniejszego opis zagadnienia (10).

Wróćmy teraz do Postgresql i zastanówmy się jak zwalczyć opisane powyżej problemy wydajnościowe. W ekstremalnym podejściu możemy wyłączyć buforowanie obiektów PreparedStatement w puli połączeń, ale oczywiście tracimy wtedy wszystkie ich zalety. Z drugiej strony ustalanie progu dla każdego niebezpiecznego polecenia przygotowanego byłoby niepraktyczne, gdyż aby pobrać je z puli, musielibyśmy znać jego treść, która najczęściej generowana jest przez Hiberante. Poza tym „zaśmieciłibyśmy” kod naszej aplikacji. Kolejnym wyjściem byłoby podzielenie aplikacji na moduły tak, by w jednych włączyć buforowanie a w innych wyłączyć. Trudno powiedzieć jak często taki podział jest możliwy, ale myślę, że rzadko. Moim zdaniem najlepsze rozwiązanie polega na odpowiednim formułowaniu zapytań. Zamiast zmiennych wiązanych użyjemy literałów w predykatkach opartych na kolumnach o wypaczonych rozkładach wartości. Dla przykładu, zamiast kodu

```
session.createQuery("from Entry e where e.isDeleted = :isDel")
    .setBoolean( „isDel”, isDeleted )
    .list();
```

wywołamy

```
if ( isDeleted )
{
    session.createQuery("from Entry e where e.isDeleted = TRUE").list();
}
else
{
    session.createQuery("from Entry e where e.isDeleted = FALSE").list();
}
```

Otrzymujemy w ten sposób większą liczbę poleceń przygotowanych, ale mamy pewność, że ich plany są wydajne. Rozwiązuje to co prawda problem pierwszych dwóch przypadków, ale wciąż pozostaje aktualny problem braku przycinania partycji. Jeśli nie możemy przenieść ich do odosobnionego modułu i wyłączyć w nim buforowania obiektów PreparedStatement, to możemy spróbować dodawać do nich predykaty zawierające literały, które wskażą optymalizatorowi dokładnie partycje które nas interesują. Zamiast

```
session.createQuery("from Entry e where e.created > :start and e.created <:end")
    .setDate( „start”, start )
    .setDate( „end”, end )
    .list();
```

piszemy:

```
String limits = { „, and e.created > DATE '2007-01-01'” } ... itd.
```

```
String bottomLimit = getBottomPartition( start );
String topLimit = getTopPartition( end );
```

```
session.createQuery("from Entry e where e.created > :start and e.created <:end"
    + bottomLimit + topLimit )
    .setDate( „start”, start )
    .setDate( „end”, end )
    .list();
```

Rozwiązanie to posiada dwie poważne wady: potencjalnie dużą liczbę możliwych kombinacji ograniczeń, która może doprowadzić do pojawienia się znacznej ilości (liczba ) rzadko wykorzystywanych zapytań. Druga wada polega na przeniesieniu wiedzy o sposobie partycjonowania tabeli z bazy do aplikacji, co pociąga za sobą konieczność aktualizacji aplikacji przy zmianie schematu partycjonowania – np. przy operacjach usunięcia bądź dodania partycji.

Kolejnym komplikacją, która niweluje zalety poleceń prekompilowanych są zapytania dynamiczne. Są one odporne na buforowanie właśnie dlatego, że konkretny zestaw warunków wybranych przez użytkownika powtarza się niezbyt często, zatem skompilowane przez narzędzie Hibernate polecenie SQL będzie za każdym razem inne. Wszystko zależy tutaj od tego na ile niepowtarzalne są owe zapytania. Jeśli bowiem na przykładowej stronie użytkownik ma do dyspozycji n – wartości dla pewnego pola, które może dowolnie wybierać - to zapytanie tego typu możemy sformułować jako :

```
from Entry e where e.email in ( :email1, :email2, .. :emailN )
```

i po ustawieniu argumentów, które wybrał użytkownik, pozostałe wypełnić wartością NULL. Jest to tylko jedna ze „sztuczek” pozwalających na ograniczenie dynamiczności zapytania, a tym samym zwiększenie możliwości zastosowania buforowania poleceń przygotowanych. Z drugiej strony – zapytania bardzo dynamiczne same w sobie nie stanowią problemu, gdyż najczęściej nie zostaną wywołane wymagana przez sterownik JDBC Postgresql liczbę razy (w takich właśnie sytuacjach przydaje się domyślne ustawienie PGStatement.getPrepareThreshold).

Reasumując – zapytania przygotowane są mechanizmem, który pozwala na zmniejszenie obciążenia procesora serwera baz danych poprzez ograniczenie ilości rozbiorów składniowych, przepisywań oraz optymalizacji planu wykonania, które musi wykonać. Nie dzieje się to za darmo, gdyż za zwiększoną wydajność płacimy pamięcią potrzebną na buforowanie obiektów PreparedStatement po stronie aplikacji oraz samych planów – po stronie serwera. Poza tym musimy dbać o odpowiednie sformułowania zapytań dotyczących: kolumn o nierównomiernym rozkładzie wartości oraz tabel partycjonowanych. Nawet gdy z powyższymi nie mamy do czynienia, to koniecznie należy zadać sobie pytania :

Jak często wykonywane są dane zapytania ?

Na ile statyczne / dynamiczne są dane zapytania ?

Jaka jest przeciętny czas życia połączenia w puli ?

i po odpowiedzeniu na nie zdecydować, czy opisywany mechanizm może przyspieszyć naszą aplikację.

Bibliografia:

1. <http://neilconway.org/talks/executor.pdf>
2. <http://wikis.sun.com/display/DBonSolaris/BuildingPostgreSQL#BuildingPostgreSQL-Adding%2F%7B%7Breadline%7D%7Dfunctionality>
3. <http://www.postgresql.org/docs/8.3/static/plpgsql-implementation.html>
4. [http://www.postgres.cz/index.php/Automatic\\_execution\\_plan\\_caching\\_in\\_PL/pgSQL](http://www.postgres.cz/index.php/Automatic_execution_plan_caching_in_PL/pgSQL)
5. <http://www.postgresql.org/docs/8.4/static/dynamic-trace.html>
6. <http://www.mail-archive.com/pgsql-performance@postgresql.org/msg18868.html>
7. <http://jdbc.postgresql.org/documentation/83/server-prepare.html>
8. [http://asktom.oracle.com/pls/asktom/f?p=100:11:0:::P11\\_QUESTION\\_ID:2588723819082](http://asktom.oracle.com/pls/asktom/f?p=100:11:0:::P11_QUESTION_ID:2588723819082)
9. <http://www.princeton.edu/~unix/Solaris/troubleshoot/dtrace.html>
10. <http://www.pythian.com/news/867/stabilize-oracle-10gs-bind-peeking-behaviour-by-cutting-histograms/>
11. <http://www.postgresql.org/docs/8.4/interactive/rules.html>
12. <http://sourceware.org/systemtap/>
13. <http://www.virtualbox.org/>
14. <http://www.opensolaris.com/>
15. <http://www.postgresql.org/docs/8.4/interactive/ddl-partitioning.html>
16. <http://optimizermagic.blogspot.com/2007/12/why-are-there-more-cursors-in-11g-for.html>