Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Damian Klata
Nr albumu: 209472

Implementacja dynamicznych widoków systemowych w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr hab. Krzysztofa Stencla
Instytut Informatyki

Wrzesień 2008

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Niniejsza praca opisuje implementację dynamicznych widoków systemowych w systemie zarządzania obiektową bazą danych LoXiM. System rozwijany jest pod opieką promotora pracy dr hab. Krzysztofa Stencla na Wydziale Matematyki Informatyki i Mechaniki Uniwersytetu Warszawskiego w oparciu o podejście stosowe przedstawione w monografii Kazimierza Subiety.

Dynamiczne widoki systemowe służą do śledzenia działania bazy danych od wykorzystania pamięci, zasobów dyskowych po obserwowanie procesów biorących udział w wykonywaniu danego zapytania. Prostota tego mechanizmu i szeroki zakres zastosowań tworzą dość silne narzędzie do optymalizacji pracy bazy danych.

Słowa kluczowe

LoXiM, system zarządzania bazą danych, widoki

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2. DATABASE MANAGMENT
H.2.3. Languages

D. Software
D.1. PROGRAMMING TECHNIQUES
D.1.5. Object-oriented Programming

Tytuł pracy w języku angielskim

An implementation of a dynamic system views in LoXiM system

Spis treści

Wprowadzenie

Niniejsza praca opisuje implementację dynamicznych widoków systemowych w systemie zarządzania obiektową bazą danych LoXiM. System rozwijany jest pod obieką promotora pracy dr hab. Krzysztofa Stencla na Wydziale Matematyki Informatyki i Mechaniki Uniwersytetu Warszawskiego w oparciu o podejście stosowe przedstawione w monografii Kazimierza Subiety.

Dynamiczne widoki systemowe służą do śledzenia działania bazy danych od wykorzystania pamięci, zasobów dyskowych po obserwowanie procesów biorących udział w wykonywaniu danego zapytania. Prostota tego mechanizmu i szeroki zakres zastosowań tworzą dość silne narzędzie do optymalizacji pracy bazy danych.

Mechanizm ma spełniać następujące kryteria:

1. przezroczysty dla zapytań
2. prosty w rozszerzaniu o nowe widoki systemowe
3. łatwy do wpięcia w pozostałe moduły systemowe

Rozdział 1 zawiera podstawowe pojęcia, które występują w dalszej części pracy. W rozdziale 2 przedstawione zostały wybrane widoki systemowe w bazie danych Oracle. W sposób praktyczny, poparty licznymi skryptami, wykazana zostanie użyteczność i szerokie spektrum zastosowań widoków. Wyjaśnione zostanie jak mogą one pomóc w podniesieniu wydajności, współbieżności bazy danych, zmniejszając obciążenie operacji dyskowych. Rozdział 3 skupia się zastosowanym rozwiązaniu w systemie LoXiM. Opis implementacja widoków systemowych jest poparty licznymi diagramami. Umieszczona została też dyskusja nad innymi możliwymi rozwiązaniami, a także pomysły na rozbudowanie obecnego zbioru widoków. Rozdział 4 prezentuje krok po kroku jak napisać nowy statystyczny widok systemowy. Jego głównym zadaniem jest praktyczne wyjaśnienie tworzenia widoków. Przeznaczony jest głównie dla deweloperów bazy danych LoXiM.

Rozdział 1

Podstawowe pojęcia

Dynamiczny widok systemowy - jest to obiekt istniejący tylko w pamięci systemu zarządzania bazą danych przedstawiający wybrane, wewnętrzne elementy działania tego systemu. Często w pracy będę pomijał słowo dynamiczny i wykorzystywał nazwę widok systemowy.

SBA (Stack Based Approach) - podejście stosowe, jest to sposób przetwarzania danych, w którym zakres zmiennych i dane wejściowe zależą od stanu stosu. Na przykład każda zmienna odwołuje się to danych położonych wyżej na stosie i oznakowanych nazwą tej zmiennej. Pozwala to łatwo zarządzać zakresem i przetwarzaniem danych poprzez wstawanie i pobieranie środowisk ze stosu.

SBQL (Stack Based Query Language)- jest nowoczesnym językiem zapytań wykorzystującym SBA. SBQL pracuje na hierarchicznych bazach danych (struktura jest podobna do XML) i realizuje założenie kompozycji, co oznacza, że dane części zapytania są niezależne. Jest to wielka zaleta w porównaniu do SQL, która pozwala pisać i analizować części zapytań i dopiero na końcu składać je w całość. SBQL może być używany jako język programowania.

Moduł składu - inaczej moduł "store" odpowiedzialny za zapis i odczyt danych z pamięci dyskowej. Zarządza między innymi buforami danych oraz mapą logicznych identyfikatorów.

Zakleszczenie - ang. deadlock - pojęcie określające sytuację, w której dwa zasoby próbują uzyskać w tym samym momencie dostęp do danego obiektu/ów. Oba założyły blokadę i oba czekają na wzajemne zwolnienie tych danych.

Rozdział 2

Wybrane dynamiczne widoki systemowe w Oracle

System zarządzania bazą danych Oracle został stworzony przez korporację Oracle Corporation ponad ćwierć wieku temu. Jest to światowy lider w branży bazodanowej. Obejmuje on 48,6 procent światowego rynku względem raportu Gartnera z 2007 roku. Dobra znajomość metod optymalizacji zapytań i technik strojenia jest istotną częścią w zarządzaniu bazą danych.

W rozdziale umieściłem szereg skryptów i objaśnień do nich przedstawiających jak i do czego można je wykorzystać. Pozwoli zrozumieć to istotę i ważność widoków systemowych.

2.1. Kursory

Kursory są jednym z najczęściej wykorzystywanych obiektów oferowanych przez Oracle. Zwracane są podczas wykonywania zapytań na bazie. Przechowują one wyniki zapytania w postaci listy wierszy, po której możemy przechodzić tylko w jednym kierunku. Odpowiednie dostosowanie parametrów bazy danych może znacząco podnieść jej współbieżność. Przykładowo jeśli z bazy korzysta tysiąc użytkowników, a maksymalną otwartą liczbę kursorów ustawimy na dziesięć, to czas reakcji na żądania pozostałych użytkowników będzie potwornie długi. Jeśli maszyna, na której pracuje baza ma wystarczającą moc, to podniesienie tego parametru powinno pomóc. Wykorzystane widoki:

- v$sesstat - widok wyświetla statystyki dla danej sesji. Aby znaleźć nazwę powiązaną z danym numerem statystyki (STATISTIC#), należy odpytać widok v$statname. Wartość danej statystyki przechowywana jest w polu value. Numer statystyki może się zmieniać w kolejnych wydaniach Oracle, stąd statystyki należy szukać po jej nazwie.
- v$statname - widok wiążący numer statystyki z jej nazwą.
- v$session - każda sesja ma jeden wiersz w tym widoku. Wiersz zawiera podstawowe informacje związane z daną sesją: użytkownik, status, adres ostatniego zapytania, czas rozpoczęcia, itp.
- v$parameter - widok przechowujący informacje o parametrach bazy danych: nazwę, wartość, typ, modyfikowalność, domyślność.
- v$bgprocess - wyświetla informacje na temat procesów działających w tle (procesy zapisujące dane, archiwalne). Każdy wiersz zawiera nazwę procesu, opis, status.

2.1.1. TotalCursorOpen.sql

Poniższy skrypt wyświetla liczbę otwartych kursorów pogrupowaną ze względu na użytkownika i maszynę, z której dany kursor został otwarty. Dzięki temu można śledzić obciążenie generowane przez danych użytkowników. W zapytaniu użyto następujących widoków: v$sesstat, v$statname, v$session. Wiążą one sesje z danymi jej statystykami oraz statystyki z ich nazwami. Praktycznie zawsze będziemy wiązać statystyki z widokiem ich nazw, ponieważ nazwa statystyki nie będzie zmieniania w kolejnych wydaniach Oracle. Liczbę otwartych kursorów uzyskujemy ze statystyki o nazwie 'opened cursors current'.

```
SELECT SUM(a.VALUE) total_cur,
       AVG(a.VALUE) avg_cur,
       MAX(a.VALUE) max_cur,
       s.username user,
       s.machine
  FROM v$sesstat a, v$statname b, v$session s
 WHERE a.statistic# = b.statistic#
   AND s.sid = a.sid
   AND b.NAME = 'opened cursors current'
 GROUP BY s.username, s.machine
 ORDER BY 1 DESC;
```

	TOTAL_CUR	AVG_CUR	MAX_CUR	USER	MACHINE
1	3	0,375	2		DAMIANK2
2	3	1,5	2	SYS	WORKGROUP/DAMIANK2

2.1.2. TuningOpenCursors.sql

Skrypt wyświetlający największą dotychczasową ilość jednocześnie otwartych kursorów oraz ograniczenie górne na liczbę otwartych kursorów ustanowione przez parametry bazy danych. Jeśli liczba otwartych kursorów jest bliska wartości maksymalnej to powinniśmy w miarę możliwości wartość maksymalną zwiększyć. Dane zwracane przez poniższe zapytanie mówi nam o obciążeniu współbieżnością bazy danych. W zapytaniu użyłem widoków: v$sesstat, v$statname b, v$parameter. Pierwsze dwa, aby zsumować statystyki otwartych kursorów ('opened cursors current'). Widok v$parameter użyty został, by pobrać maksymalną liczbę otwartych kursorów. Odpowiada za to parametr 'open_cursors'.

```
SELECT MAX(a.VALUE) AS highest_open_cur, p.VALUE AS max_open_cur
  FROM v$sesstat a, v$statname b, v$parameter p
 WHERE a.statistic# = b.statistic#
   AND b.NAME = 'opened cursors current'
   AND p.NAME = 'open_cursors'
 GROUP BY p.VALUE;
```

	HIGHEST_OPEN_CUR	MAX_OPEN_CUR
1	3	300

2.1.3. SessionsCachedCursors.sql

Skrypt zlicza ilość zbuforowanych kursorów. Baza danych może zbuforować kursor w celu ponownego jego wykorzystania w przyszłości. Przykładowo jeśli użytkownik wykonuje dwa

razy to samo zapytanie, to za pierwszym razem prawdopodobnie wykonany zostanie pełny cykl parsowania i wykonania zapytania. Za drugim razem baza danych skorzysta z wcześniej zapamiętanych danych i wykona tylko część czynności obsługujących zapytanie. Skróci to znacząco czas jego wykonania. Im większy współczynnik buforowania kursorów tym lepiej. W celu wyciągnięcia informacji o współczynniku trafień w bufor kursorów oraz liczby parsowań kursorów posłużyłem się widokami v$sesstat, v$sesstat używając statystyk 'session cursor cache hit' i 'parse count (total)'. Dany wiersz zapytania wyświetla informacje dla jednej sesji.

```
SELECT cach.VALUE cache_hits,
       prs.VALUE all_parses,
       prs.VALUE - cach.VALUE sess_cur_cache_not_used
  FROM v$sesstat cach, v$sesstat prs, v$statname nm1, v$statname nm2
 WHERE cach.statistic# = nm1.statistic#
   AND nm1.NAME = 'session cursor cache hits'
   AND prs.statistic# = nm2.statistic#
   AND nm2.NAME = 'parse count (total)'
   AND prs.sid = cach.sid;
```

	CACHE_HITS	ALL_PARSES	SESS_CUR_CACHE_NOT_USED
1	0	0	0
2	0	66	66
3	0	1852	1852
4	0	132	132
5	0	4	4
6	0	9	9
7	0	62	62
8	0	0	0
9	0	0	0
10	0	0	0

2.2. Sesje i zapytania

Sesje Oraclowe wykorzystywane są do komunikacji użytkownika z bazą danych. Baza danych przyjmując połączenie użytkownika tworzy dla niego oddzielną sesję, która w wątku obsłuży jego zlecenia. Śledząc zapytania sesji obciążające mocno bazę danych poprzez operację odczytu i zapisu możemy znacznie łatwiej wykryć przyczynę błędu, zanalizować ponownie zapytanie, zoptymalizować je lub odpowiednio dostosować parametry bazy danych. Wykorzystane widoki:

- v$sesstat - widok wyświetla statystyki dla danej sesji. Aby znaleźć nazwę powiązaną z danym numerem statystyki (STATISTIC#), należy odpytać widok v$statname. Wartość danej statystyki przechowywana jest w polu value. Numer statystyki może się zmieniać w kolejnych wydaniach Oracle, stąd statystyki należy szukać po jej nazwie.

- v$statname - widok wiążący numer statystyki z jej nazwą.

- v$session - każda sesja ma jeden wiersz w tym widoku. Wiersz zawiera podstawowe informacje związane z daną sesją: użytkownik, status, adres ostatniego zapytania, czas rozpoczęcia, itp.

- v$sess_io - widok zawiera statystyki odczytu/zapisu danych na dysk. Statystyki zbierane są dla każdej sesji.
- v$session_longops - widok wyświetla status różnych operacji, które trwają dłużej niż 6 sekund. Te operacje uwzględniają wiele funkcji tworzenia kopii i odzyskiwania danych, zbierania statystyk, wykonania zapytania i wiele innych. Każde wydanie Oracle poszerza zakres widzialnych w tym widoku danych. Aby monitorować proces wykonywania zapytania, trzeba używać optymalizatora kosztowego (cost-based) i włączyć opcję TIMED_STATISTICS lub SQL_TRACE.

2.2.1. Sessions.sql

Prosty skrypt wyświetlający aktualne sesję użytkowników. W poniższym zapytaniu wyświetlane są identyfikator sesji, czas zalogowania użytkownika, status sesji. Status mówi czy dana sesja przetwarza aktualnie jakieś zapytanie. Skrypt pozwala śledzić aktywność użytkowników bazy danych w danym momencie. W przykładzie korzystam z jednego prostego widoku sesji v$session.

```
SELECT sid,
       to_char(logon_time, 'MM-DD-YYYY HH24:MI') logon_time,
       username,
       TYPE,
       status,
       sql_address
  FROM v$session
 WHERE username IS NOT NULL
```

	SID	LOGON_TIME	USERNAME	TYPE	STATUS	SQL_ADDRESS
1	8	08-18-2008 09:21	SYS	USER	INACTIVE	0
2	0	08-18-2008 09:23	SYS	USER	ACTIVE	172F558C

2.2.2. SessionsStats.sql

Skrypt wyświetla statystyki operacji odczytu i zapisu dla danej sesji. Zwracane parametry mówią nam, które sesje mocno obciążają pamięć dyskową. Te najbardziej obciążające możemy zanalizować i poprawić zapytania wykonywane na nich. Jeśli jest bardzo dużo odczytanych fizycznie stron z dysku w porównaniu do odczytu wszystkich stron danych, to być może powinniśmy zwiększyć pamięć bazy danych odpowiedzialną za buforowanie stron pamięci. W zapytaniu korzystam z dwóch widoków: v$sess_io, v$session. Pierwszy w celu wyciągnięcia potrzebnych informacji o operacjach odczytu/zapisu. Drugi po to, by wykluczyć z wyników sesje nie posiadające użytkowników.

```
SELECT sess_io.sid,
       sess_io.block_gets,
       sess_io.consistent_gets cons_get,
       sess_io.physical_reads,
       sess_io.block_changes
  FROM v$sess_io sess_io, v$session sesion
 WHERE sesion.sid = sess_io.sid
   AND sesion.username IS NOT NULL
```

	SID	BLOCK_GETS	CONS_GETS	PHYSICAL_READS	BLOCK_CHANGES
1	8	0	1056	87	0
2	10	0	319	18	0

2.2.3. LongOperations.sql

Długie operacje (long operations), to operacje powstające podczas wykonywania zapytań mocno obciążających system. Jedno złe zoptymalizowane zapytanie potrafi spowolnić działanie całej bazy danych. Może ono potrzebować nawet kilka gigabajtów pamięci dyskowej, w celu posortowania lub złączenia danych. Podany skrypt pomaga wykryć niezoptymalizowane zapytania lub inne "ciężkie" operacje. Poprawiając otrzymane wyniki zwiększymy wydajność bazy danych oraz zmniejszymy jej obciążenie. W poniższym przykładzie korzystam z widoku v$session_longops wyświetlającego długie operacje oraz z widoku v$session pozwalającego ograniczyć mi wyniki do odpowiednich modułów.

```
SELECT time_remaining, sofar, elapsed_seconds
  FROM v$session_longops l, v$session s
 WHERE l.sid = s.sid
   AND l.serial# = s.serial#
   AND s.module = 'long_proc'
```

	TIME_REMAINING	SOFAR	ELAPSED_SECONDS
1	205	51000	11

2.2.4. TopIOSessions.sql

Skrypt zwracający "najcięższe" sesje dla systemu zarządzania bazą danych posortowane po współczynniku tot_io_pct. "Ciężkie" sesje wykorzystują dużą liczbę operacji odczytu i zapisu na dysku. Wskaźnik tot_io_pct mówi nam w jakim stopniu (procencie) dana sesja wykorzystuje zasoby systemu. Daje nam to podpowiedź na jakie sesje należy zwrócić uwagę w celu poprawienia wydajności. W zapytaniu użyłem następujących widoków: v$statname, v$sesstat, v$session, v$bgprocess. Pierwsze dwa służą do wyciągnięcia odpowiednich statystyk dla sesji: liczby fizycznych odczytów ('physical reads') i zapisów ('physical writes'), liczby bezpośrednich fizycznych odczytów ('physical writes direct') i zapisów ('physical reads direct') oraz liczby odczytów i zapisów w obiektach typu lob ('physical writes direct (lob)', 'physical reads direct (lob)'). Otrzymane statystyki sumujemy dla każdej z sesji oraz dla całości. Dzięki temu możemy określić procentowy udział poszczególnych sesji w ilości operacji odczytu/zapisu. Widoki v$session i v$bgprocess pozwalają nam określić nazwę użytkownika lub procesu danej sesji. Jeśli sesja była wywołana przez użytkownika, to pole v$session..username będzie nie puste. W przeciwnym przypadku korzystając z drugiego widoku pokażemy nazwę odpowiedniego procesu.

```
SELECT sid, username, round(100 * total_user_io / total_io, 2) tot_io_pct
  FROM (SELECT b.sid sid,
               nvl(b.username, p.NAME) username,
               SUM(VALUE) total_user_io
          FROM sys.v$statname  c,
               sys.v$sesstat   a,
               sys.v$session   b,
               sys.v$bgprocess p
```

```
        WHERE a.statistic# = c.statistic#
          AND p.paddr(+) = b.paddr
          AND b.sid = a.sid
          AND c.NAME IN
              ('physical reads', 'physical writes',
               'physical writes direct', 'physical reads direct',
               'physical writes direct (lob)',
               'physical reads direct (lob)')
        GROUP BY b.sid, nvl(b.username, p.NAME)),
      (SELECT SUM(VALUE) total_io
         FROM sys.v$statname c, sys.v$sesstat a
        WHERE a.statistic# = c.statistic#
          AND c.NAME IN
              ('physical reads', 'physical writes',
               'physical writes direct', 'physical reads direct',
               'physical writes direct (lob)',
               'physical reads direct (lob)'))
ORDER BY 3 DESC;
```

	SID	USERNAME	TOT_IO_PCT
1	2	DBW0	72,03
2	5	SMON	16,29
3	9	QMN0	4,28
4	8	SYS	2,53
5	15	TEST	1,4
6	3	LGWR	1,34
7	12	TEST	1,08
8	10	SYS	0,55
9	14	TEST	0,44
10	6	RECO	0,03
11	7	CJQ0	0,03
12	1	PMON	0
13	4	CKPT	0

2.3. Blokady

Blokady są to obiekty zakładane przez bazę danych na dane wiersze lub tabele w celu wyłącznego lub współdzielonego korzystania z nich. Pozwalają one zachować spójność danych w środowisku współbieżnym. Jest to bardzo silne narzędzie niepozbawione wad. Najczęściej spotykanym problemem związanym z blokadami są zakleszczenia (deadlock). Zakleszczenie następuje wtedy, gdy dwa procesy próbują uzyskać dostęp do zablokowanych przez siebie wzajemnie obiektów i czekają na ich zwolnienie. Sposoby wykrywania zakleszczeń i ich usuwania opisywane są praktycznie w każdym artykule, książce poświęconej współbieżnym systemom. Nie wszystkie zakleszczenia jesteśmy w stanie usunąć automatycznie. Do tego celu posłużyłby nam podany niżej skrypt. Jeśli gdzieś w systemie wystąpi zakleszczenie wystarczy wtedy odczytać jakie dwie sesji chcą jednocześnie uzyskać dostęp do tych samych zasobów i przeanalizować zapytania, którymi te sesje się posłużyły. Wykorzystane widoki:

- v$session - każda sesja ma jeden wiersz w tym widoku. Wiersz zawiera podstawowe

informacje związane z daną sesją: użytkownik, status, adres ostatniego zapytania, czas rozpoczęcia, itp.

- v$lock - widok pokazuje blokady aktualnie założone przez bazę danych Oracle oraz oczekujące żądania na blokadę lub zatrzask.
- dba_objects - opisuje wszystkie obiekty w bazie danych. Każdy wiersz zawiera informacje na temat danego obiektu: nazwę, typ, czas utworzenia, czas modyfikacji, status, itp.

2.3.1. ShowLocks.sql

Skrypt wyświetla wszystkie zakleszczenia jakie aktualnie występują w systemie. W opisie wiersza znajduje się informacja jakie dwa procesy (sesje i maszyna) dokonały blokad na jakim obiekcie. W poniższym zapytaniu skorzystałem z następujących widoków: v$lock, v$session, dba_objects. Widok v$lock pozwolił wykryć zakleszczenie dwóch sesji. Dokładniejsze informacje o sesji związanej z daną blokadę wyciągnąłem z widoku v$session. Nazwa blokowanego obiektu umieszczona jej w widoku dba_objects. Uzyskujemy ją podając odpowiedni identyfikator obiektu.

```
SELECT s1.username || '@' || s1.machine || ' ( SID=' || s1.sid ||
       ' )  is blocking ' || s2.username || '@' || s2.machine ||
       ' ( SID=' || s2.sid || ' ) ' AS blocking_status,
       do.object_name obj_name,
       s2.row_wait_obj# as obj,
       s2.row_wait_file# as f,
       s2.row_wait_block# as blk,
       s2.row_wait_row# as row,
       dbms_rowid.rowid_create(1,
                               s2.row_wait_obj#,
                               s2.row_wait_file#,
                               s2.row_wait_block#,
                               s2.row_wait_row#) as rid
  FROM v$lock l1, v$session s1, v$lock l2, v$session s2, dba_objects do
 WHERE s1.sid = l1.sid
   AND s2.sid = l2.sid
   AND l1.BLOCK = 1
   AND l2.request > 0
   AND l1.id1 = l2.id1
   AND l2.id2 = l2.id2
   AND s2.row_wait_obj# = do.object_id;
```

	BLOCKING_STATUS	OBJ_NAME	OBJ	F	BLK	ROW	RID
1	TEST@WORKGROUP /DAMIANK2 (SID=13) is blocking TEST@WORKGROUP/ DAMIANK2 (SID=17)	TSTLOCK	35600	1	75522	0	AAAIsQ AABAAA ScCAAA

Rozdział 3

Implementacja dynamicznych widoków systemowych w LoXiM

3.1. Opis rozwiązania

Podczas pisania implementacji dynamicznych widoków systemowych w LoXiM przyjąłem następujące założenia:

- Przezroczystość
- Prosty w rozszerzaniu o nowe obiekty systemowe
- Łatwy do wpięcia w pozostałe moduły systemu

Przezroczystość dla zapytań oznacza, że odwoływanie się do obiektów systemowych nie różni się niczym składniowo od zapytań do zwykłych obiektów. System po nazwie obiektu, do którego się chcemy uzyskać dostęp, określa czy jest to dynamiczny obiekt systemowy czy zwykły zapisany na dysku.

Prostota ma polegać na tym, że napisanie nowego obiektu systemowego nie będzie wymagało od programisty bazy danych zbyt wiele wysiłku projektowego. Nowe obiekty to w rzeczywistości kawałki kodu na sztywno umieszczone w źródłach bazy danych. Podstawą rozwiązania są jasno określone reguły i niezbyt skomplikowany wzorzec.

Mechanizm ma mieć możliwość wglądu w pozostałe części systemu, aby badać stan pamięci, ilość operacji odczytu lub zapisu, itp. Stąd istnieje konieczność, by inne części systemu zbierały potrzebne dla nas informacje i umieszczały je poprzez pewien moduł statystyczny w określonym miejscu, tak abyśmy mogli je w odpowiedniej chwili przeczytać.

Implementacja widoków systemowych, mimo że wydaje się dość prostym pomysłem, nie była łatwa. Zmodyfikowałem ponad 70 klas już istniejących elementów systemu. Jest to dość szeroki moduł zakresem obejmujący praktycznie cały system. Główny problemem było zaprojektowanie abstrakcyjnego wzorca projektowe tak, aby można go było wielokrotnie używać. Przezroczystość uzyskałem modyfikując gramatykę języka używanego w LoXiM. Wszelkie odwołania do widoków odbywają się poprzez podanie nazwy widoku poprzedzonego prefiksem %. Przykład:

```
deref(%AllViews)
```

Podane wyżej zapytanie wyświetla zawartość widoku AllViews rozwijając rekurencyjnie wskaźniki do obiektów.

Zmiana gramatyki sprawiła, że od strony użytkowej stosowanie widoków systemowych niczym nie różni się od używania innych obiektów. Można korzystać praktycznie w pełni z potęgi oferowanego przez system LoXiM języka zapytań. Niestety z uwagi, że są to obiekty istniejące tylko w pamięci, pojawiają się dwa ograniczenia:

- Nie możemy zapisywać danych do tych obiektów z zapytania
- Nie powinniśmy zapamiętywać wskaźników do obiektów widoku systemowego w innych obiektach

Podane wyżej ograniczenia wynikają z tego, że referencje do obiektów nieustannie się zmieniają. Logiczny identyfikator zmienia się podczas każdego odwołania do obiektu widoku systemowego.

Kolejnym pomysłem umożliwiającym spełnienie przezroczystości jest odpowiednie wykorzystanie logicznych identyfikatorów. Logiczny identyfikator jest to obiekt w systemie LoXiM wskazujący na dany obiekt. W kolejnych bitach przeważnie jest zapisany plik, strona i przesunięcie obiektu na stronie. Stąd mając jedną liczbę 32-bitową jesteśmy w stanie dokładnie sprecyzować położenie obiektu na dysku. W tym celu zostało napisanych kilka procedur tłumaczących adres logiczny (identyfikator) na fizyczny, czyli dokładny adres obiektu na dysku. W moim rozwiązaniu przyjąłem, że pewna pula adresów będzie niedostępna dla zwykłych obiektów. Obiekty, których identyfikator logiczny ma "zapalone" osiem pierwszych bitów to obiekty widoków systemowych (bity 31-25). Wszystkie pozostałe kombinacje bitów tworzą identyfikator normalnych obiektów, czyli obiektów przechowywanych fizycznie na dysku. W sumie zabrałem około 2^{24} wartości identyfikatora logicznego z puli 2^{32}. Niewielkim kosztem i zmniejszeniem nieznacznie puli dostępnych identyfikatorów logicznych stworzyłem mechanizm dostępu do obiektów widoków systemowych praktycznie niewidoczny dla większości modułów. Moduły odpowiedzialne za parsowanie, wykonanie i optymalizację zapytań nie zdają sobie sprawy na jakiego typu obiektach operują. Rozgraniczenie następuje dopiero w module "Store", czyli module składu danych.

Skład danych otrzymując konkretne żądania odczytu lub zapisu danego obiektu stwierdza na podstawie identyfikatora czy wyciągnąć dany obiekt z pliku czy z danego widoku systemowego. Jeśli z widoku systemowego to przekazuje zlecenie odczytu do klasy SystemViews zarządzającej widokami.

Zarządzanie widokami systemowymi jest scentralizowane w jednej klasie SystemViews. Klasa zawiera mapę widoków systemowych, które są rozróżnianie po nazwie danego widoku. Zatem nazwa widoku musi być unikalna w obrębie całego systemu. Klasa zawiera także mechanizm przydzielania nowych identyfikatorów logicznych obiektów powstających przy tworzeniu i aktualizacji widoków systemowych. Mechanizm polega na tym, że w tablicy trzymamy informację o wykorzystanych lub wolnych identyfikatorach. Jest to tablica bitowa, a każdy bit zapalony mówi nam, że identyfikator o odpowiednim numerze jest wykorzystany. Chcąc wykorzystać nowy identyfikator szukam pierwszego wolnego identyfikatora zaczynając szukanie od ostatnio użytego. Znaleziony identyfikator logiczny wiążę z nowym obiektem.

Tworzenie nowego widoku systemowego powstanie poprzez napisanie klasy dziedziczącej po klasie SystemView. Do zaimplementowania mamy kilka metod z nadklasy odpowiedzialnych za inicjalizację, finalizację, odświeżanie widoku oraz pobieranie podobiektu tego widoku systemowego. Tak skonstruowaną klasę rejestrujemy w klasie zarządzającej widokami SystemViews poprzez metodę registerView nadając przy tym unikalną nazwę widokowi. Poprzez tę nazwę uzyskujemy dostęp do widoku z poziomu zapytań.

Inicjalizacja widoków systemowych zarejestrowanych w menedżerze widoków odbywa się przy starcie systemu. Odświeżanie natomiast przy każdorazowym odwołaniu do tego widoku.

Zatem jeśli ktoś nie korzysta z tego mechanizmu, to nie będzie on wpływał na stan pamięciowy bazy danych. Warto podkreślić, że podobiekty należące do widoku systemowego nie są tworzone od razu przy inicjalizacji, ale dopiero w momencie pierwszego odwołania do nich.

W systemie utworzyłem osiem widoków systemowych. Podzielone są na dwie grupy. Pierwsza grupa to widoki proste:

- Information - wyświetla informacje o bazie danych (wersję, nazwę)
- AllViews - wyświetla zbiór wszystkich zarejestrowanych widoków systemowych
- Counter - licznik zwiększający się przy każdym odwołaniu do niego

Druga grupa to widoki statystyczne:

- Sessions - przechowuje informacje o sesjach
- Store - przechowuje statystyki modułu "store" odpowiedzialnego bezpośrednio za zapis i odczyt obiektów z pamięci dyskowej
- Configs - przechowuje informacje o konfiguracji bazy danych
- Transactions - zawiera statystyki transakcji
- Queries - zawiera informacje o wykonywanych zapytaniach

Widoki statystyczne jest to podgrupa widoków systemowych. Głównym ich celem jest prezentowanie statystyk zbieranych z innych modułów systemu. Bazują one na klasie SystemStatsView dziedziczącej po SystemView. Klasa ta jest ściśle związana z klasą AllStats przechowującą statystyki danych obszarów systemu zarządzania bazą danych. Nowy statystyczny widok systemowy powstaje poprzez utworzenie i zarejestrowanie klasy SystemStatsView podając w konstruktorze nazwę statystyki, którą chcemy prezentować w tymże widoku.

Statystyki jest to moduł systemu odpowiedzialny za zbieranie informacji z innych części systemu. Podstawową klasą, na której bazują wszystkie inne klasy statystyk, jest SystemStats. Klasa przechowuje mapę wartości związanych z daną statystyką. Wartość identyfikujemy poprzez jej unikalną nazwę. Przyjmuje jeden z czterech rodzajów dostępnych typów:

- string - tekstowa
- int - liczbowa całkowita
- double - liczbowa rzeczywista, zmiennoprzecinkowa
- SystemStats - obiektowa, przechowująca podstatystyki

Mając wyżej wymienione typy, można z prostych klas skonstruować wielopoziomowe, rozbudowane statystyki o strukturze drzewiastej. Przykładem może być statystyka sesji, w której bazowa statystyka zawiera zbiór statystyk poszczególnych sesji. Statystyka sesji natomiast przechowuje w swoich wartościach informacje na temat ilości zapisanych i odczytanych danych.

Prostota oraz duża abstrakcyjność wyżej przedstawionego wzorca klasy statystyki i jej wartości umożliwia implementacje konkretnych statystyk w prosty i szybki sposób. W statystykach przyjąłem konwencję, że zapis informacji do statystyki realizuję poprzez wystawione metody. Jeśli chcemy zapisać pojedynczą wartość, przykładowo czas rozpoczęcia sesji, to w klasie SessionStats wystawiamy metodę setStartTime. Metoda ta wywołując ogólną metodę setStringStats zapisuje tekst do mapy wartości:

```
void SessionStats::setStartTime(string value) {
setStringStats("START_TIME", value);
}
```

Bardziej zaawansowane modyfikacje wartości statystyk wykonuję w złożonych metodach. W jednej metodzie mogę dokonywać trudniejszych obliczeń oraz zapisywać wiele wartości statystycznych. Upraszcza to nam korzystanie z takiej klasy, a komplikuje jedynie jej implementację. Przykład:

```
void SessionStats::addDiskPageReads(int count) {
diskPageReads += count;
setIntStats("DISK_PAGE_READS", diskPageReads);
double hit = 0.0;
if (pageReads > 0) {
hit = 100 * ((1.0 * (pageReads - diskPageReads))
/ pageReads);
}
setDoubleStats("PAGE_READS_HIT", hit);
}
```

W systemie istnieje specjalna statystyka AllStats. Charakterystyczną jej cechą jest to, że mamy do niej dostęp poprzez metodę statyczną getHandle. Statystyka AllStats zawiera wszystkie inne statystyki systemowe. Jeśli chcemy korzystać ze statystyk w innych częściach systemu zarządzania bazą danych LoXiM, to musimy użyć tej właśnie klasy. Przykład:

```
LoximServer::LoximSession::LoximSession(
LoximServer *server,
AbstractSocket *socket,
ErrorConsole *err_cons
)
{
this->server = server;
this->socket = socket;
this->layer0 = 0;
this->err_cons = err_cons;
this->id = get_new_id();
this->shutting_down = false;
this->user_data = NULL;
this->error = 0;
this->qEx = new QueryExecutor(this);
this->KAthread = new KeepAliveThread(this, err_cons);
this->worker = new Worker(this, err_cons);
pthread_mutex_init(&send_mutex, 0);
error = 0;
for (int i = 0; i < 20; i++)
salt[i] = (char)(rand()%256-128);
stats = new SessionStats();
stats->setId(this->id);

/* Utworzenie numeru sesji */
```

```
stringstream ss;
ss << "SESSION_" << (this->id);S
ss >> sessionid;

AllStats::getHandle()->getSessionsStats()->
addSessionStats(sessionid, stats);
}
```

Ogólność statystyk bardzo przydała się w statystycznych widokach systemowych. Nie trzeba było pisać nowej klasy widoku dla każdej ze statystyk. Uniwersalny mechanizm zawarty w klasie SystemStatsView utworzy widok obsługujący daną statystykę na podstawie jej nazwy. Klasa SystemStatsView zaktualizuje statystyki oraz przetłumaczy wartości statystyczne na podobiekty o odpowiednich typach.

Odświeżanie statystyki następuje w dwóch momentach. Pierwszy kiedy odwołujemy się do widoku związanego z tą statystyką. Drugi następuje w momencie zmiany wartości mierzonej. Przykładem może być pomiar ilości odwołań do stron w pamięci bazy danych. Jeśli dokonujemy odczytu danego obiektu z dysku, to jest on wrzucany do pamięci. W tym momencie uruchamiana jest metoda ze statystyk inkrementująca licznik odczytanych stron oraz jest odświeżany wskaźnik trafień odczytu.

Implementacja mechanizmu statystyk przysporzyła wielu trudności. Pisząc statystyki sesji zmuszony byłem zmodyfikować kilkanaście klas systemu tak, aby otrzymać wymagane informacje. Przykładowo chcąc zliczyć ilość operacji zapisu danych dla danej sesji musiałem dodać do metod zapisu obiektów modułu składu (store) informację kto wywołał daną metodę. Takiej informacji nie było, a okazała się bardzo przydatna. W wielu innych obszarach systemu występowały nie raz takie sytuacje. Tak więc cała trudność w dodawaniu nowych statystyk leży w niepełnym zasobie informacji, których potrzebujemy w konkretnych rejonach systemu. Tworzenie klas statystyk, widoków systemowych związanych z nimi oraz odpowiednia aktualizacja ich jest już stosunkowo prosta.

Zaprezentowane rozwiązanie doskonale spełnia przyjęte założenia. Przezroczystość uzyskana za pomocą zmiany gramatyki języka zapytań systemu LoXiM oraz wydzielenie pewnej puli adresów logicznych na potrzeby widoków systemowych. Prostota w rozszerzaniu o nowe obiekty systemowe rozwiązana została dzięki stworzeniu uniwersalnego mechanizmu statystycznych widoków systemowych oraz dużej ogólności statystyk. Łatwość wpięcia w pozostałe moduły systemu zrealizowałem dzięki dostępnej z każdego miejsca systemu klasie AllStats. Rozwiązanie jest proste i szybkie. Zawiera wiele ogólnych wzorców projektowych umożliwiających rozbudowę mechanizmu widoków i statystyk o nowe elementy względem jasno określonego schematu.

3.2. Inne rozwiązania

Podczas projektowania podanego w poprzedniej sekcji rozwiązania rozważałem inne sposoby zrealizowania widoków systemowych. Istniejący projekt ma jedną wadę. Jest on sztywno włączony w źródła bazy danych. Chcąc dodać nowy widok systemowy musimy każdorazowo modyfikować istniejącą strukturę klas systemu. Stąd narodził się pomysł, aby uniezależnić widoki od kodu źródłowego. Idea była taka, aby dodanie widoku systemowego zrealizować tak jak zwykłe widoki. Tworzenie widoku systemowego odbywałoby się poprzez zapytanie, a wynik zapisywany na dysku. Dzięki takiemu podejściu otrzymujemy łatwo rozszerzalny mechanizm z możliwością dostosowania go do potrzeb konkretnych użytkowników. Podczas

dalszej analizy tego pomysłu pojawiły się wątpliwości. Czy na pewno jest potrzebna taka ogólność? Jak zaimplementować opisane rozwiązanie?

Potrzeba tak dużej ogólności wydaje się mała. Trzeba zauważyć, że widoki są dość zaawansowanym mechanizmem śledzenia systemu bazy danych. Pomijając sprawy implementacyjne, użytkownik sam musiałby określać wskaźnik trafień bufora odczytu. Użytkownik stosując mniej lub bardziej skomplikowane wzory musiałby zapytaniem wyliczać odpowiednie wartości, co nie jest proste. Zatem upraszczając dodawanie nowych widoków od strony źródeł bazy danych, komplikujemy ich dodawanie od strony zapytania. W praktyce oznacza to, że stopień skomplikowania samego widoku nie zmienia się. Zmienia się tylko miejsce tej komplikacji. Łatwiej kontroluje się źródła programu, wyspecjalizowane klasy niż bardzo ogólny mechanizm i jego wywołania. Jest to argument za istniejącym rozwiązaniem

Drugi problem stanowią kwestie implementacyjne. W jakiś sposób trzeba informować użytkowników o zaistniałych zdarzeniach systemu, na przykład zapis strony dyskowej. Jak taki widok miałby zostać poinformowany? Jednym z rozwiązań jest wystawienie metod systemowych do poboru tych informacji poprzez rozszerzenie gramatyki. Metody te czerpałyby informacje z zapisanych gdzieś w systemie informacjach. Widzimy, że i w tym rozwiązaniu dane statystyczne muszą być przechowywane gdzieś w pamięci. Każde rozwiązanie, które będzie rozważać niestety musi takie dane przechowywać w systemie. Różnica polega na tym jak będziemy je udostępniać. Podany wyżej pomysł ma tą wadę, że każda nowa statystyka będzie wymaga rozszerzenia gramatyki. Zaimplementowane rozwiązanie wymusza jedynie dodanie nowej klasy bez ingerencji w język zapytań. Język zapytań powinien pozostać jak najmniejszy. Zbędne komplikacje mogą bardziej zaszkodzić niż pomóc. Stąd to rozwiązanie odrzuciłem.

Podsumowując dane statystyczne muszą być przechowywane w pamięci, nie chcemy modyfikować w dużym stopniu gramatyki języka, nowo dodane statystyki powinny być chociaż w półautomatyczny sposób włączane w system. Te warunki spełnia obecne rozwiązanie.

3.3. Proponowane rozszerzenia

W systemie zarządzania bazą danych LoXiM zostało zaimplementowanych kilka podstawowych widoków systemowych śledzących działanie transakcji, sesji, operacji dyskowych modułu składu, konfiguracji oraz wykonywanych zapytań. Warto dodać następujące widoki:

- widok blokad systemowych
- widok wykorzystanej pamięci systemowej
- widok użytkowników
- widok uprawnień użytkowników
- widok obiektów głównych bazy danych (roots)
- widok zajętości plików bazy danych

Widok blokad systemowych odpowiedzialny byłby za wyświetlanie aktualnie utworzonych blokad na obiektach wykorzystywanych w danym zapytaniu sesji. Taki widok umożliwiłby ewentualne wykrycie zakleszczeń (deadlock). Zakleszczenia są jednym z poważniejszych problemów w programowaniu wielowątkowych. Sposoby ich wykrywania i usuwania opisywane są praktycznie w każdym artykule, książce poświęconej współbieżnym systemom. Nie wszystkie zakleszczenia jesteśmy w stanie usunąć automatycznie. Do tego celu posłużyłby nam ten widok. Jeśli takie wystąpi wystarczyłoby odczytać jakie dwie sesji chcą jednocześnie uzyskać

dostęp do tych samych zasobów i anulować jedną z nich. Następnie należałoby przeanalizować zapytania, którymi te sesje się posłużyły w celu wykluczenia ponownego wystąpienia zakleszczenia.

Widok wykorzystywanej pamięci systemowej jest pomocny w dostrajaniu bazy danych. Jeśli pamięci jest za mało to następuje efekt migotania. Baza zaczyna zbyt często zrzucać dane na dysk. Potrafi to znacząco spowolnić działanie całego systemu. Taki widok byłby w stanie pokazać w jakim stopni i w jakich obszarach pamięć bazy danych jest wykorzystywana najbardziej.

Widok użytkowników prezentowałby listę wszystkich użytkowników bazy danych. Ułatwiłby zarządzanie użytkownikami, ich dostępem do bazy danych i śledzeniem wykonywanych przez nich operacji.

Widok uprawnień użytkowników stanowi dobre wyjście do narzędzi kontrolujących dostęp do danych obiektów. Łatwo moglibyśmy zobaczyć kto jakie ma prawa i ewentualnie je dodawać lub usuwać poprzez wbudowane polecenia.

Widok obiektów głównych bazy danych (roots) jest podstawą do wielu narzędzi operujących na strukturze bazy danych. Obecnie brak jest polecenia wyświetlającego wszystkie węzły główne w systemie. Moim zdaniem jest to duży minus. Mając podany wyżej widok otworzylibyśmy drogę między innymi do narzędzia porównującego obiekty między dwoma bazami. Przykładowo w bazie Oracle mamy dwa sposoby na porównanie struktury bazy danych. Pierwszy to poprzez specjalną metodę porównującą dwa obiekty ze sobą, drugi poprzez otworzenie kursorów na odpowiednich tabelach systemowych przechowujących metadane. Średni czas porównywania dwóch schematów 10 GB baz danych poprzez metodę składowaną zajmuje około dwóch godzin, poprzez kursory około dziesięciu minut [1]. Czas takiego porównania korzystając z widoków systemowych oracle jest o rząd mniejszy.

Widok zajętości plików bazy danych obrazowałby procent wykorzystania poszczególnych plików danych przez zapisane obiekty systemu. Dzięki niemu moglibyśmy zdecydować czy powiększyć dany plik, czy utworzyć nowy lub odchudzić już istniejący.

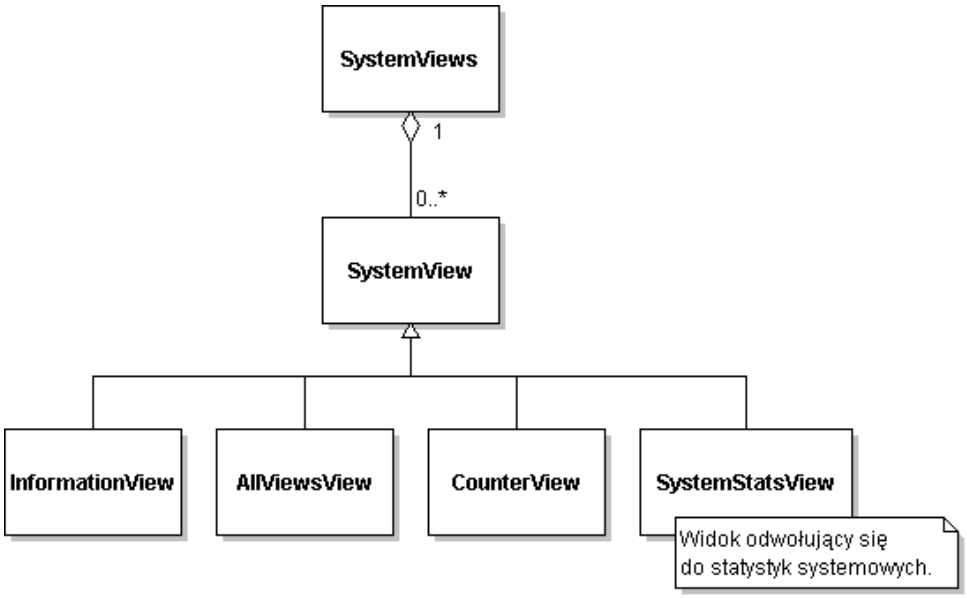
3.4. Diagramy klas

3.4.1. Widoki systemowe

Klasy widoków systemowych

Relacje między klasami związanymi z widokami systemowymi. Klasa SystemViews jest kontenerem zawierającym obiekty klasy SystemView.

[1] Czasy uzyskane z przeprowadzonych przeze mnie testów. Mogą się różnić na innych maszynach oraz innych strukturach schematów.



SystemViews

Klasa zarządzająca widokami systemowymi. Zawiera:

- mapOfViews - mapa widoków wiążąca nazwę widoku z danym widokiem. Nowe widoki rejestrujemy za pomocą metody registerView.
- tableOfUsedId - tabela bitowa zajętości identyfikatorów logicznych związanych z podobiektami widoków systemowych. N-ty bit "zapalony" oznacza, że identyfikator o adresie FF000000 + N jest wykorzystywany przez dany obiekt. Tworzenie nowego identyfikatora odbywa się w metodzie createNextId. Aby zwolnić niepotrzebny identyfikator używamy releaseId.
- getObject - metoda zwracająca obiekt powiązany z danym identyfikatorem. Obiekty są tworzone i zarządzane w klasach dziedziczących po klasie SystemView.

SystemViews
unsigned int tableOfUsedId[0x80000] unsigned int usedIdCount unsigned int currentId ObjectPointer* emptyObject map<const char*, SystemView*, strCmp> mapOfViews
SystemViews() virtual ~SystemViews() vector<int>* getItems(TransactionID* tid) vector<int>* getItems(TransactionID* tid, const char* name) virtual int getObject(TransactionID* tid, LogicalID* lid, AccessMode mode, ObjectPointer*& object) int createNextId(LogicalID*& id) void releaseID(LogicalID* id) void registerView(const char* name, SystemView* view)

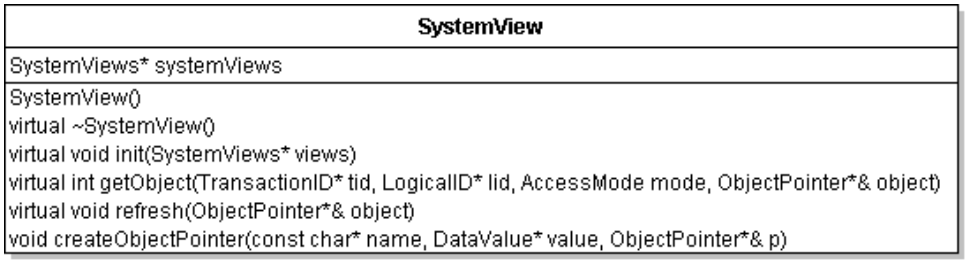
SystemView

Główna klasa widoków. Po niej dziedziczą wszystkie widoki systemowe. Metody:

- init - w tej metodzie podklasy powinny inicjalizować wszystkie potrzebne zmienne.

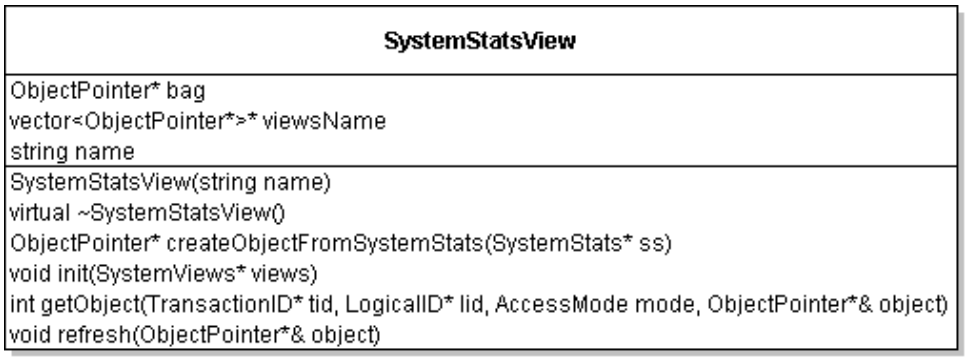
- refresh - odświeżenie widoku

- getObject - widok poprzez tą metodę zwraca obiekt związany z danym identyfikatorem logicznym



SystemStatsView

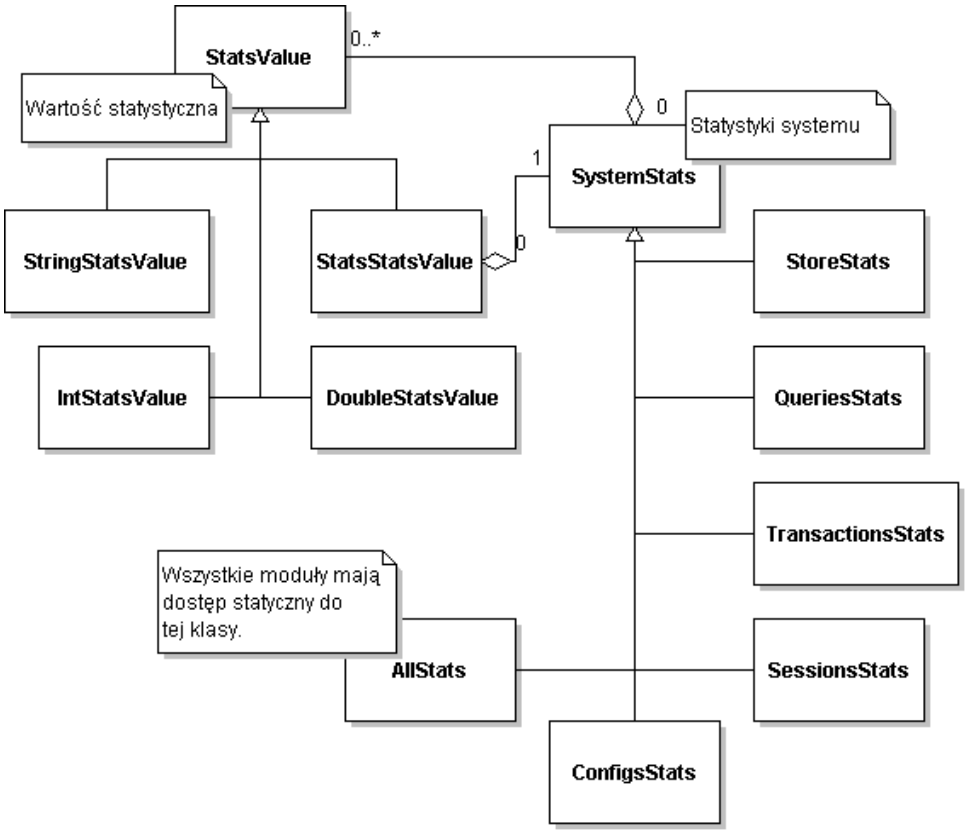
Klasa obsługująca statystyczne widoki systemowe. Jest ona ściśle związana z klasą AllStats przechowującej statystyki danych obszarów systemu zarządzania bazą danych. Nowy statystyczny widok systemowy tworzymy poprzez utworzenie i zarejestrowanie klasy SystemStatsView podając w konstruktorze nazwę statystyki, którą chcemy prezentować w tymże widoku. Zawiera mechanizm tłumaczenia wartości statystycznych na obiekty odpowiednich typów.



3.4.2. Statystyki

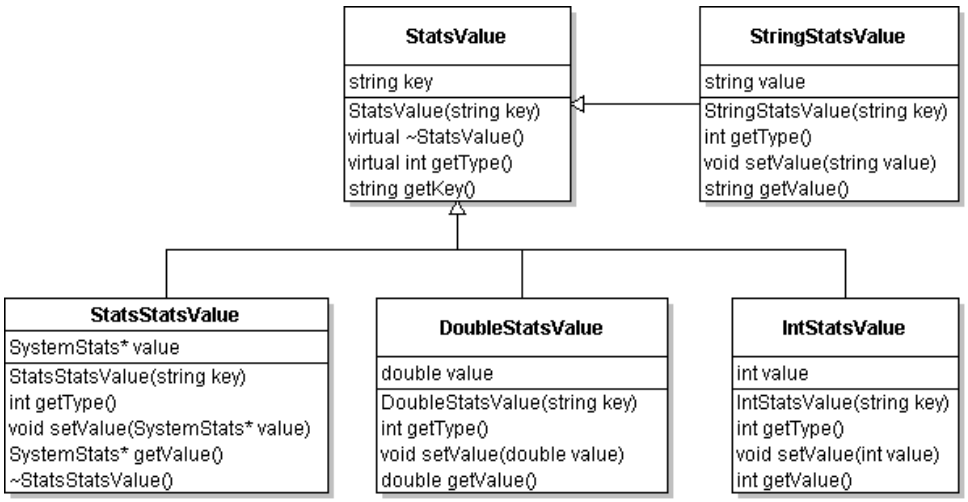
Klasy statystyk

Powiązanie między klasami związanymi ze statystykami systemowymi. Głównym ich elementem jest klasa SystemStats.



Wartości statystyczne

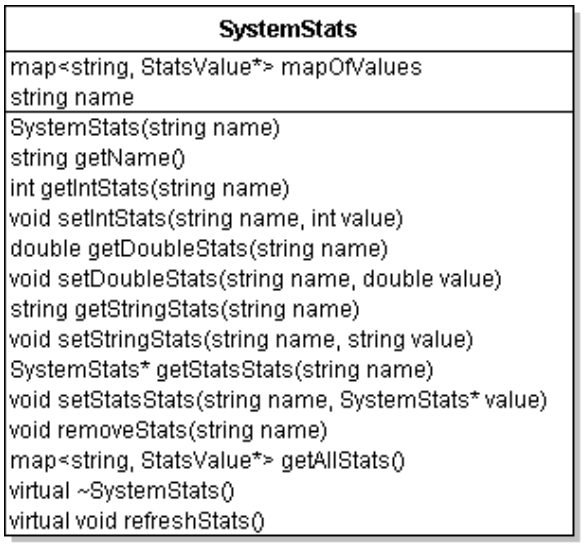
Relacje pomiędzy wartościami statystycznymi. Klasa wartości zawiera swoją unikalną w obrębie statystyki nazwę. Wspólną cechą podklas jest metoda getType, która pozwala w szybki sposób zidentyfikować typ wartości. Każdy rodzaj wartości zwraca inną wartości liczbową w metodzie getType. We wszystkich podklasach są metody ustawiające i pobierające wartość z klasy dostosowane do przechowywanego w nich typu.



SystemStats

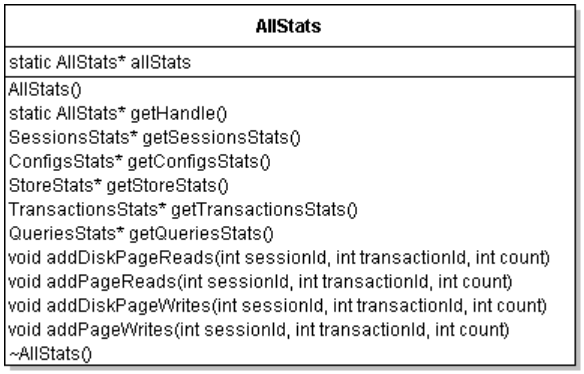
Podstawowa klas statystyk. Zawiera metody:

- getName - zwraca nazwę statystyki
- getIntStats, setIntStats, ... - metody obsługujące wartości statystyk. SystemStats przechowuje dane statystyczne w mapie wiążącej nazwę z klasą wartości.
- getAllStats - zwraca wszystkie wartości statystyczne w postaci mapy
- refreshStats - odświeża daną statystykę od strony widoków systemowych



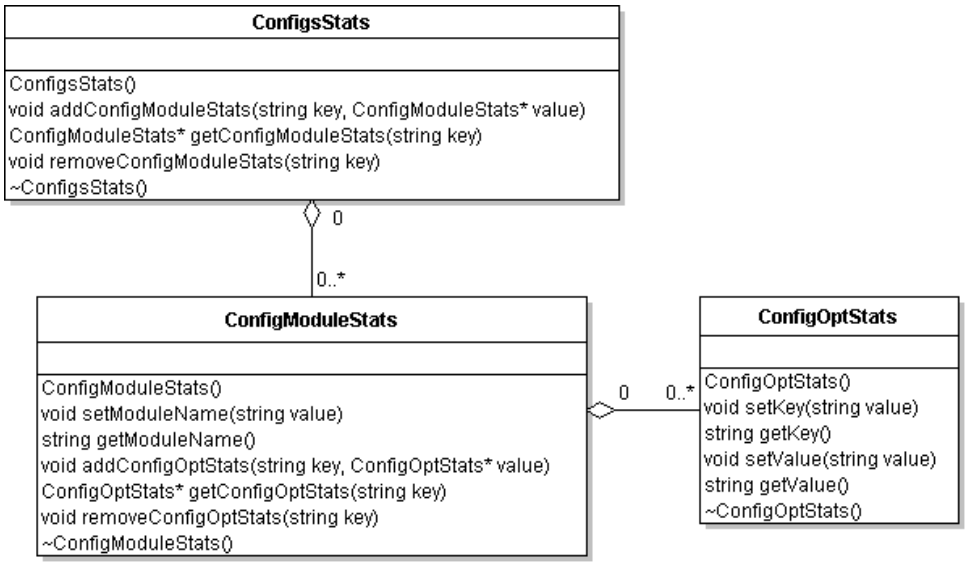
AllStats

Klasa stanowiąca "most" między statystykami, a innymi częściami systemu. Korzystając z tej klasy system zapisuje dane w odpowiednich statystykach. Dostęp do niej jest zrealizowany metodą statyczną getHandle. Poszczególne statystyki pobieramy metodami getConfigStats, getTransactionStats, ... Ponadto w klasie znajdują się dodatkowe metody pomocnicze, wspomagające zapisywanie podobnych informacji w różnych statystykach.



ConfigStats

Statystyki parametrów bazy danych. Prezentuje wszystkie parametry zapisane w konfiguracji plikowej bazy danych podzielone względem modułów. Pozwalają zdalnie sprawdzić ustawienie startowe systemu zarządzania bazą danych LoXiM.

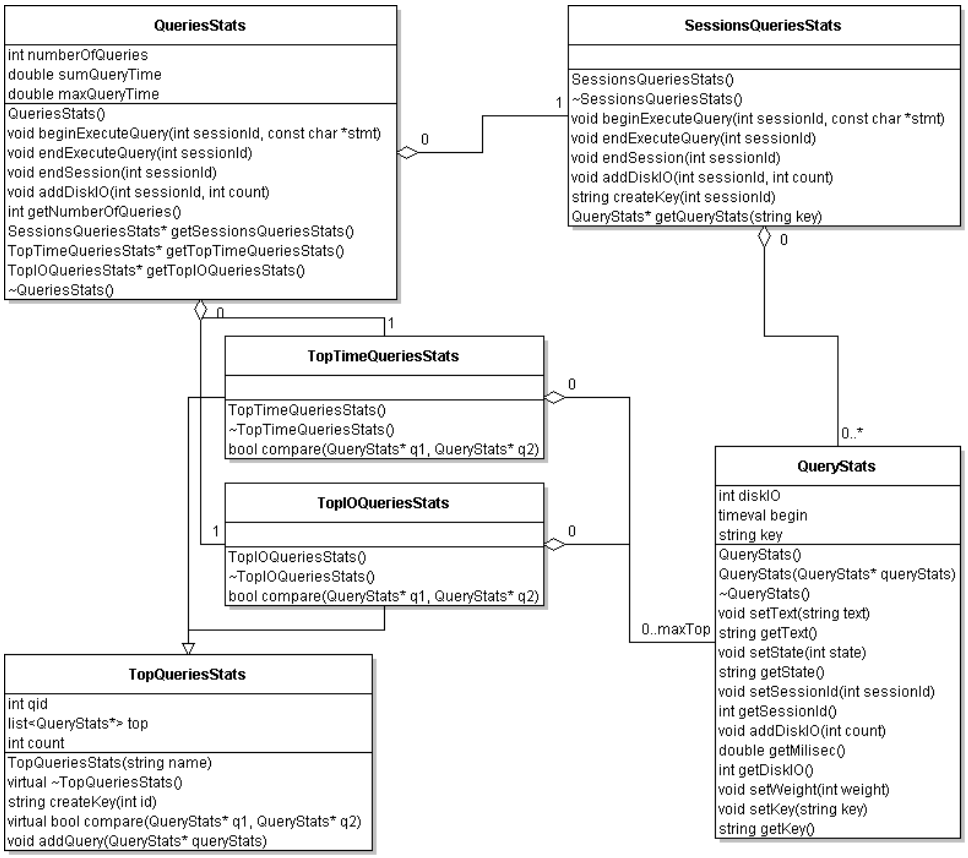


QueriesStats

Statystyki zapytań prezentują:

- Czasy trwania zapytania dla danego połączenia
- Ilość zapytań
- Średni czas zapytania
- Maksymalny czas zapytania
- 10 ostatnich najbardziej obciążających zapytań wg czasu wykonania
- 10 ostatnich najbardziej obciążających zapytań wg ilości operacji odczytu i zapisu

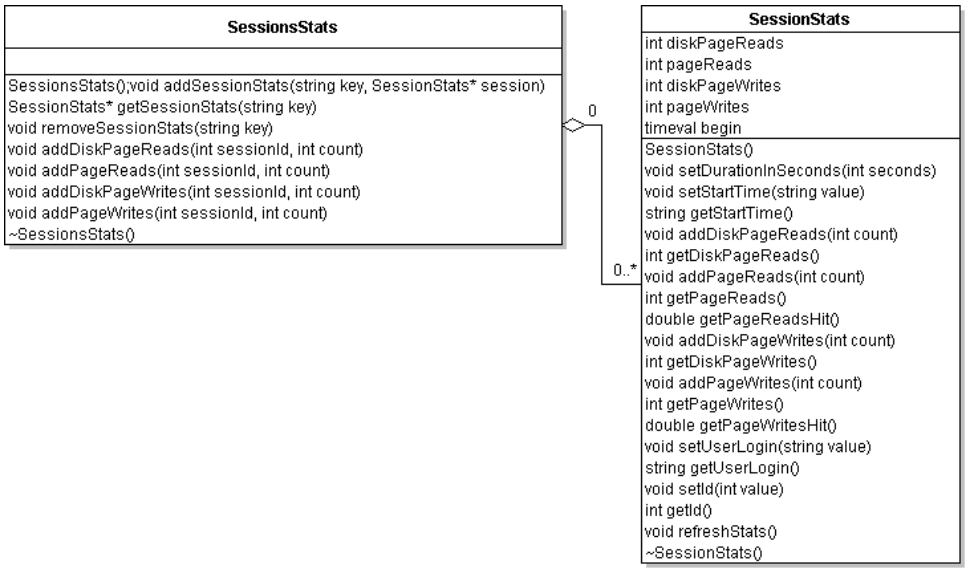
Przydatne bardzo w analizie obciążeń systemu przez konkretne zapytania. Jesteśmy w stanie łatwo wykryć, zanalizować i poprawić najbardziej obciążające zapytanie. Mocne narzędzie potrafiące znacząco pomóc w odciążeniu zużywanych zasobów systemu.



SessionsStats

Statystyki połączeń pomagają śledzić aktywność użytkowników i powodowane przez nich obciążenia. Przechowują następują informacje:

- Godzina rozpoczęcia
- Czas trwania
- Ilość operacji we/wy
- Login użytkownika



StoreStats

Statystyki modułu składu. Zawierają:

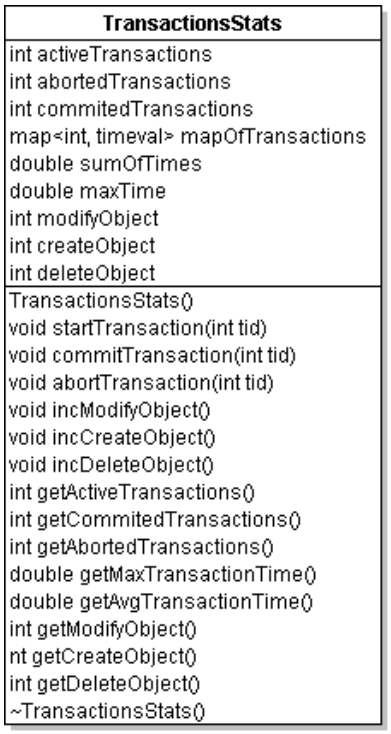
- Ilość operacji odczytu i zapisu
- Maksymalny czas odczytu danych
- Średni czas odczytu danych

Analizując podane statystyki możemy stwierdzić czy zmodyfikować wielkość bufora stron w celu poprawienia wskaźnika trafień stron. Wskaźnik trafień odczytu jest to procent stron odczytanych z pamięci bez konieczności ładowania ich z dysku do liczby wszystkich odczytanych stron. Im większy tym pamięć dyskowa jest mniej obciążona, a zatem baza danych pracuje wydajniej. Zbyt mały wskaźnik oznacza, że bufor jest niewystarczający i występuje efekt migotania. Efekt ten polega na tym, że potrzebne dane dyskowe nieustannie są odczytywane do pamięci i za chwilę z niej usuwane z powodu niedostatecznej pamięci. W rezultacie mocno wzrasta ilość operacji dyskowych.

TransactionsStats

Statystyki transakcji wspomagają analizę wykonania zapytań w danej transakcji. Dzięki nim możemy sprawdzić ostatnio wykonywane operacje na bazie danych. W skład statystyki wchodzą:

- Ilość transakcji
- Maksymalny czas trwania transakcji
- Średni czas trwania transakcji
- Liczba wywołań poszczególnych metod (tworzenia, usuwanie i modyfikacji obiektu)



3.5. Diagramy sekwencji

Diagramy przedstawiają poszczególne procesy jakie występują podczas wykonywania zapytań. Przykładowe zapytanie, na którym oparłem schematy wygląda następująco:

```
deref(%Store)
```

Zapytanie żąda wyświetlenia wartości widoku systemowego Store rozwijając je rekurencyjnie. Przetwarzanie zapytanie dzieli się na dwa etapy:

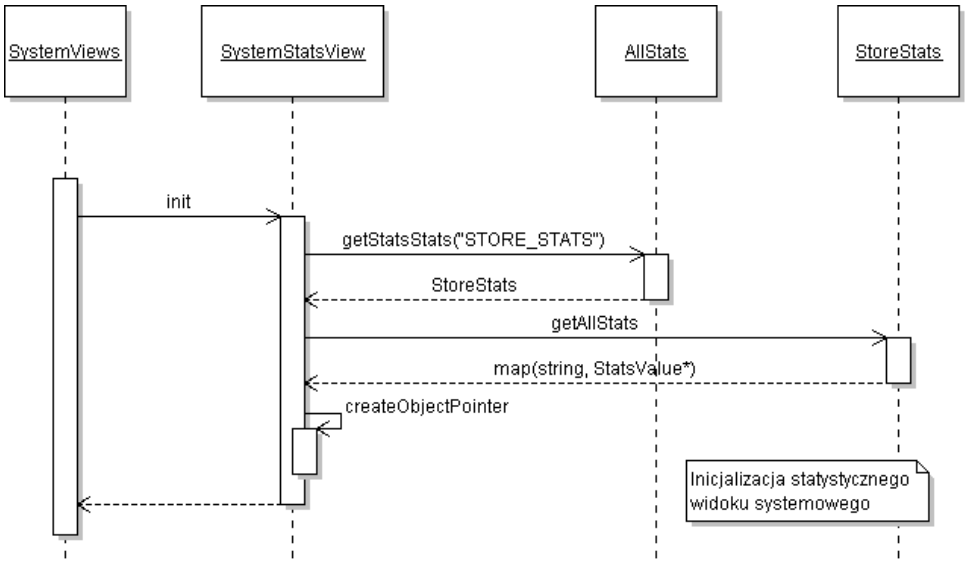
- Parsujemy i wykonujemy zapytanie prosząc o identyfikatory obiektów głównych widoku o danej nazwie (getItems)

- Pobieramy kolejne obiekty o danych identyfikatorach logicznych przechodząc drzewo parsowania rekurencyjnie i składając wyniki w odpowiednią całość.

Pierwszy diagram obrazuje proces inicjalizacji statystycznego widoku systemowego.

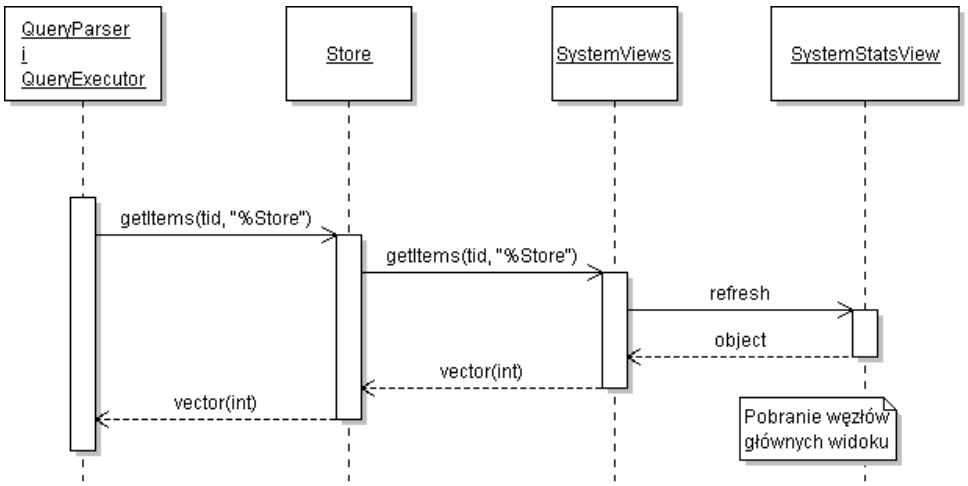
Inicjalizacja widoku systemowego

Diagram przedstawia proces inicjalizacji statystycznego widoku systemowego. Podczas inicjalizacji następuje utworzenie obiektów na podstawie danej statystyki.



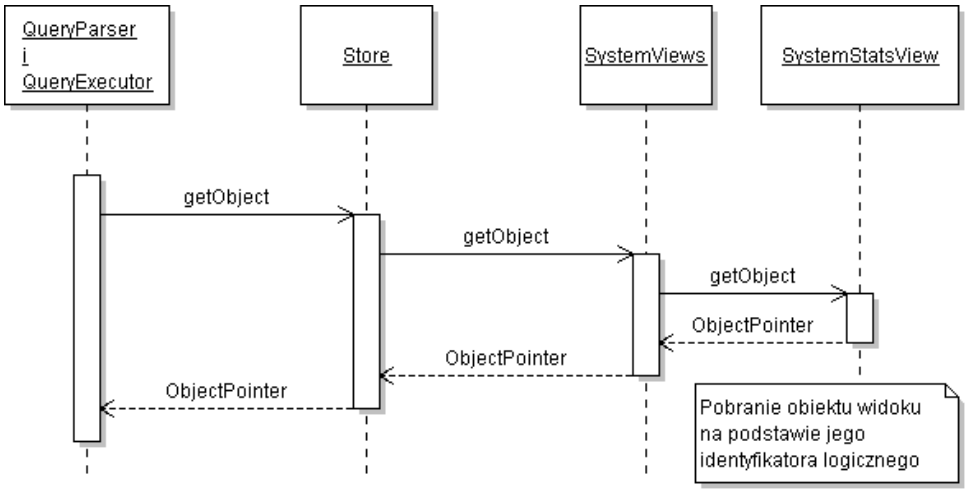
Pobranie węzłów głównych widoku systemowego

Pobieranie węzłów głównych wykorzystywane jest w procesie przetwarzania zapytania. Widok zwraca identyfikatory węzłów podobiektów leżących na najwyższym poziomie w hierarchii.



Pobranie podobiektu widoku systemowego

Pobranie podoobiektu z widoku systemowego o danym identyfikatorze logicznym. Oczywiście podany na diagramie proces ma miejsce, gdy pierwsze osiem bitów identyfikatora logicznego jest "zapalone".


QueryParser
i
QueryExecutor
Store
SystemViews
SystemStatsView
getObject
getObject
getObject
ObjectPointer
ObjectPointer
ObjectPointer
Pobranie obiektu widoku
na podstawie jego
identyfikatora logicznego

Rozdział 4

Jak dopisać nowy widok systemowy w LoXiM

W niniejszym rozdziale zostanie szczegółowo opisane tworzenie statystycznego widoku systemowego. Krok po kroku przedstawiono implementację statystyk konfiguracji bazy danych, widoku systemowego związanego z nią oraz interakcję z odpowiednimi częściami systemu. Plik konfiguracji dzieli się na moduły, a każda konfiguracja modułu składa się z wpisów klucz=wartość. Przykładowy plik konfiguracji:

```
[Store]
store_file_default=/tmp/sbdefault
store_file_map=/tmp/sbmap
[QueryParser]
optimisation=off
```

Podany plik zawiera wpisy do dwóch modułów Store i QueryParser. Store zawiera dwie opcje store_file_default i store_file_map, QueryParser tylko optimisation.

Naszym celem będzie przejrzyste odwzorowanie tego pliku na widok systemowy, tak aby można było wykonywać na nim przydatne zapytania (znajdź konfigurację o danej nazwie). Przede wszystkim należy przenieść hierarchiczność z pliku na widok i statystyki. Zaprojektujmy taki podział statystyk:

- Statystyka konfiguracji - główny obiekt, składa się ze statystyk modułów
- Statystyka modułu - zawiera opcję danego modułu oraz jego nazwę
- Statystyka opcji - przechowuje odwzorowanie klucz=wartosć

Jeśli w ten sposób napiszemy statystyki, to widok systemowy automatycznie odwzoruje hierarchiczność i wartości statystyczne na dane podobiekty.

4.1. Statystyki - nagłówek

Tworzenie statystycznego widoku systemowego zaczniemy od statystyk konfiguracji. W tym celu należy w pakiecie SystemStats utworzyć plik nagłówkowy ConfigStats.h. W pliku zdefiniowane będą trzy klasy ConfigsStats (statystyka konfiguracji), ConfigModuleStats (statystyka modułu), ConfigOptStats (statystyka opcji). Zwiążemy je następującymi relacjami: ConfigsStats zawiera obiekty klasy ConfigModuleStats oraz ConfigModuleStats zawiera obiekty klasy ConfigOptStats.

4.1.1. Krok 1

Zdefiniujmy klasę ConfigOptStats przechowującą jedynie dwie wartości statystyczne klucz (key) i wartość (value). Wszystkie klasy statystyk muszą dziedziczyć po klasie SystemStats. Oprócz konstruktora i destruktora zdefiniujemy w klasie metody, którymi będziemy ustawiać lub odczytywać dane wartości statystyczne: setKey, getKey, setValue, getValue.

```
#ifndef CONFIGSTATS_H_
#define CONFIGSTATS_H_
#include "SystemStats.h"

using namespace std;

namespace SystemStatsLib {
/*
 * Configs stats is the simples stats that show
 * only configuration file readed from disk.
 */
class ConfigOptStats: public SystemStats {

public:
ConfigOptStats();

void setKey(string value);
string getKey();

void setValue(string value);
string getValue();

~ConfigOptStats();
};

}

#endif /* CONFIGSTATS_H_ */
```

4.1.2. Krok 2

Statystyki modułu zdefiniujemy w klasie ConfigModuleStats. Klasa zawiera zbiór statatystyk opcji oraz nazwę. Nazwę ustawiamy poprzez metodę setModuleName, odczytujemy ją przez getModuleName. Metody addConfigOptStats, removeConfigOptStats i getConfigOptStats służą kolejno do dodania nowej statystyki opcji do zbioru, usunięcia istniejącej statystyki opcji i pobranie statystyki opcji o danym kluczu.

```
class ConfigModuleStats: public SystemStats {

public:
ConfigModuleStats();

void setModuleName(string value);
```

```
string getModuleName();

void addConfigOptStats(string key,
ConfigOptStats* value);
ConfigOptStats* getConfigOptStats(string key);
void removeConfigOptStats(string key);

~ConfigModuleStats();
};
```

4.1.3. Krok 3

Ostatnią klasą jaką należy zdefiniować jest główna klasa statystyki konfiguracji, czyli ConfigsStats. Klasa jest praktycznie identyczna z ConfigModuleStats. Zawiera zbiór statystyk modułu. Metody addConfigModuleStats, removeConfigModuleStats i getConfigModuleStats służą do dodania, usunięcia i pobrania statystyki modułu o danym kluczu.

```
class ConfigsStats: public SystemStats {

public:
ConfigsStats();

void addConfigModuleStats(string key,
ConfigModuleStats* value);
ConfigModuleStats* getConfigModuleStats(string key);
void removeConfigModuleStats(string key);

~ConfigsStats();
};
```

4.1.4. Podsumowanie

Połączywszy trzy poprzednie punkty w całość otrzymujemy następujący plik nagłówkowy definiujący trzy klasy statystyk:

```
/*
 * ConfigStats.h
 *
 *  Created on: 5-lip-08
 *      Author: damianklata
 */

#ifndef CONFIGSTATS_H_
#define CONFIGSTATS_H_
#include "SystemStats.h"

using namespace std;

namespace SystemStatsLib {
/*
```

```
 * Configs stats is the simples stats that show
 * only configuration file readed from disk.
 */
class ConfigOptStats: public SystemStats {

public:
ConfigOptStats();

void setKey(string value);
string getKey();

void setValue(string value);
string getValue();

~ConfigOptStats();
};

class ConfigModuleStats: public SystemStats {

public:
ConfigModuleStats();

void setModuleName(string value);
string getModuleName();

void addConfigOptStats(string key,
ConfigOptStats* value);
ConfigOptStats* getConfigOptStats(string key);
void removeConfigOptStats(string key);

~ConfigModuleStats();
};

class ConfigsStats: public SystemStats {

public:
ConfigsStats();

void addConfigModuleStats(string key,
ConfigModuleStats* value);
ConfigModuleStats* getConfigModuleStats(string key);
void removeConfigModuleStats(string key);

~ConfigsStats();
};
}

#endif /* CONFIGSTATS_H_ */
```

4.2. Statystyki - implementacja

W tej części zajmiemy się implementacją wcześniej zdefiniowanych klas statystyk. Implementacja opiera się na wzorcach projektowych opisanych w poprzednim rozdziale. Tworzymy plik ConfigStats.cpp w pakiecie SystemStats. Jego zawartość będzie tworzyć w trzech kolejnych krokach.

4.2.1. Krok 1

Zacznijmy od implementacji statystyk opcji. W konstruktorze zawsze musimy wywołać konstruktor nadklasy z nazwą tej statystyki. Podobiekty widoku systemowego prezentujące tą statystykę otrzymają podaną nazwę. Metody zapisu i odczytu wartości statystycznych w swojej treści zawierają wywołania standartowych metod odpowiedzialnych za zapis i odczyt danych odpowiedniego typu: setStringStats, getStringStats.

```
#include "ConfigStats.h"

using namespace SystemStatsLib;

ConfigOptStats::ConfigOptStats() :
SystemStats("OPTION") {
setKey("UNKNOWN");
setValue("UNKNOWN");
}

void ConfigOptStats::setKey(string value) {
setStringStats("KEY", value);
}

string ConfigOptStats::getKey() {
return getStringStats("KEY");
}

void ConfigOptStats::setValue(string value) {
setStringStats("VALUE", value);
}

string ConfigOptStats::getValue() {
return getStringStats("VALUE");
}

ConfigOptStats::~ConfigOptStats() {
}
```

4.2.2. Krok 2

Implementacja statystyk modułu w swoich metodach zawiera standartową obsługę wartości statystycznych. Nazwę modułu zapisujemy wywołując metodę z nadklasy setStringStats, odczytujemy ją metodą getStringStats. Do obsługi zbioru statystyk służą metody:

- setStatsStats(key, value) - dodaje statystykę do zbioru o danym kluczu
- getStatsStats(key) - pobiera statystykę o danym kluczy. Zwróconą statystykę należy zrzutować na odpowiednią podklasę.
- removeStats(key) - usuwa statystykę o danym kluczu

Aby obsłużyć poprawnie dodawanie, usuwanie i odczytywanie statystyk opcji wystarczy w ciele metod addConfigOptStats, removeConfigOptStats, getConfigModuleStats użyć wyżej wymienionych metod z nadklasy.

```
ConfigModuleStats::ConfigModuleStats() :
SystemStats("MODULE") {
setModuleName("UNKNOWN");
}

void ConfigModuleStats::setModuleName(string value) {
setStringStats("MODULE_NAME", value);
}

string ConfigModuleStats::getModuleName() {
return getStringStats("MODULE_NAME");
}

void ConfigModuleStats::addConfigOptStats(string key,
ConfigOptStats* value) {
setStatsStats(key, value);
}

ConfigOptStats* ConfigModuleStats::getConfigOptStats(string key) {
return dynamic_cast<ConfigOptStats*>(getStatsStats(key));

}

void ConfigModuleStats::removeConfigOptStats(string key) {
removeStats(key);
}

ConfigModuleStats::~ConfigModuleStats() {
}
```

4.2.3. Krok 3

Została już tylko implementacja statystyki konfiguracji. Jest ona analogiczna do implementacji zbioru statystyk opcji w statystykach modułu.

```
ConfigsStats::ConfigsStats() :
SystemStats("CONFIGS") {

}
```

```
void ConfigsStats::addConfigModuleStats(string key,
ConfigModuleStats* value) {
setStatsStats(key, value);
}

ConfigModuleStats* ConfigsStats::getConfigModuleStats(
string key) {
return dynamic_cast<ConfigModuleStats*>(
getStatsStats(key));
}

void ConfigsStats::removeConfigModuleStats(string key) {
removeStats(key);
}

ConfigsStats::~ConfigsStats() {

}
```

4.2.4. Podsumowanie

Poniżej umieszczam zebrane w całość trzy poprzednie kroki. Wynikowy plik implementacji statystyki konfiguracji przedstawia się następująco:

```
#include "ConfigStats.h"

using namespace SystemStatsLib;
/* =-=-=-=-=-=-=-=-=-=- */
ConfigOptStats::ConfigOptStats() :
SystemStats("OPTION") {
setKey("UNKNOWN");
setValue("UNKNOWN");
}

void ConfigOptStats::setKey(string value) {
setStringStats("KEY", value);
}

string ConfigOptStats::getKey() {
return getStringStats("KEY");
}

void ConfigOptStats::setValue(string value) {
setStringStats("VALUE", value);
}

string ConfigOptStats::getValue() {
return getStringStats("VALUE");
```

```
}

ConfigOptStats::~ConfigOptStats() {
}

/* =-=-=-=-=-=-=-=-=-=- */

ConfigModuleStats::ConfigModuleStats() :
SystemStats("MODULE") {
setModuleName("UNKNOWN");
}

void ConfigModuleStats::setModuleName(string value) {
setStringStats("MODULE_NAME", value);
}

string ConfigModuleStats::getModuleName() {
return getStringStats("MODULE_NAME");
}

void ConfigModuleStats::addConfigOptStats(string key,
ConfigOptStats* value) {
setStatsStats(key, value);
}

ConfigOptStats* ConfigModuleStats::getConfigOptStats(string key) {
return dynamic_cast<ConfigOptStats*>(getStatsStats(key));

}

void ConfigModuleStats::removeConfigOptStats(string key) {
removeStats(key);
}

ConfigModuleStats::~ConfigModuleStats() {
}

/* =-=-=-=-=-=-=-=-=-=- */

ConfigsStats::ConfigsStats() :
SystemStats("CONFIGS") {

}

void ConfigsStats::addConfigModuleStats(string key,
ConfigModuleStats* value) {
setStatsStats(key, value);
}
```

```
ConfigModuleStats* ConfigsStats::getConfigModuleStats(
string key) {
return dynamic_cast<ConfigModuleStats*>(
getStatsStats(key));
}

void ConfigsStats::removeConfigModuleStats(string key) {
removeStats(key);
}

ConfigsStats::~ConfigsStats() {

}
```

Należy pamiętać jeszcze o dopisaniu odpowiednich wpisów dotyczących zarówno pliku nagłówkowego jak i implementacyjnego w pliku Makefile.

4.3. Statystyki AllStats

Statystyki konfiguracji musimy udostępnić dla innych modułów. Trzeba zatem zmodyfikować klasę AllStats, przez którą inne moduły komunikują się ze statystykami. Najpierw dopisujemy metodę, która zwróci statystyki konfiguracji. Modyfikujemy plik AllStats.h

```
class AllStats: public SystemStats{
protected:
static AllStats* allStats;
public:
...
ConfigsStats* getConfigsStats();
...
};
```

Następnie dopisujemy implementację metody getConfigsStats w pliku AllStats.cpp, a także inicjujemy statystyki konfiguracji w konstruktorze klasy.

```
AllStats::AllStats(): SystemStats("ALL_STATS") {
setStatsStats("SESSIONS_STATS", new SessionsStats());
setStatsStats("STORE_STATS", new StoreStats());
setStatsStats("CONFIGS_STATS", new ConfigsStats());
...
}

...

ConfigsStats* AllStats::getConfigsStats() {
return dynamic_cast<ConfigsStats*>(
getStatsStats("CONFIGS_STATS"));
}
```

4.4. Widok systemowy

Korzystając ze specjalnej klasy obsługującej statystyczne widoki systemowe dodamy widok konfiguracji do systemu. Klasa SystemStatsView zajmuje się tłuczeniem hierarchii statystyk na podobiekty danego widoku systemowego. Wystarczy, że zarejestrujemy nasz widok wywołując registerView w ciele metody init podając odpowiednie parametry. Dokładna treść znajduje się poniżej:

```
SystemViews::SystemViews()
{
...
/* In this section of code we are registering
 * all system views
 */
registerView("%Configs",
new SystemStatsView("CONFIGS_STATS"));
...

/* Init views */
map<const char*,SystemView*>::iterator iter;
for( iter = mapOfViews.begin();
iter != mapOfViews.end(); iter++ ) {
iter->second->init(this);
}
}
```

Objaśnienia:

- %Configs - jest to nazwa widoku systemowego widocznego od strony zapytań
- CONFIGS_STATS - nazwa statystyk konfiguracji znajdująca się w AllStats

4.5. Moduł konfiguracji

Ostatni krok jaki został, to "wpięcie" w moduł konfiguracji. Moduł konfiguracji podczas odczytu pliku powinien przekazać wszystkie zebrane opcje do statystyki konfiguracji. Zrealizujemy to na końcu metody inicjalizującej (init) zaraz po odczytaniu pliku.

```
...
#include "SystemStats/ConfigStats.h"
#include "SystemStats/AllStats.h"
...

using namespace SystemStatsLib;

namespace Config {

int SBQLConfig::init(string file)
{
...
```

```
/* Add config to stats */
ConfigsStats* cs = AllStats::getHandle()->
getConfigsStats();
ModuleOptions* module = config;
while (module != NULL) {
ConfOpt* opt = module->options;
ConfigModuleStats* cms =
new ConfigModuleStats();
cms->setModuleName(module->name);
while (opt != NULL) {
ConfigOptStats* cos =
new ConfigOptStats();
cos->setKey(opt->name);
cos->setValue(opt->value);
cms->addConfigOptStats(opt->name, cos);
opt = opt->nextOpt;
}
cs->addConfigModuleStats(module->name, cms);
module = module->nextMod;
}
return 0;
};
...
}
```

Rozdział 5

Posumowanie

Zaimplementowana w systemie LoXiM implementacja widoków systemowych spełnia wszystkie postawione podczas projektowania zadania. Zastosowany zbiór widoków jest faktycznie bardzo pomocny. Dokładnie oddaje stany wybranych obszarów systemu. Do pełności przedstawionego mechanizmu brakuje jeszcze kilku widoków opisanych w możliwych rozszerzeniach systemu.

Widoki systemowe w bazie danych Oracle były punktem wyjściowym dla mojej pracy. Szereg skryptów przedstawiających jak można je wykorzystać miał za zadanie wykazać ważność i sens tych obiektów. Przeniesienie ich do systemu LoXiM otworzyło drogę do wielu narzędzi analizujących obciążenia systemu, porównujących struktury baz danych. Są do narzędzia wspomagające proces strojenia bazy danych.

Dodatek A

Dołączona płyta

A.1. Zawartość dołączonej płyty

Katalog szbd zawiera pliki źródłowe ostatniej wersji systemu zarządzania bazą danych LoXiM, na której opracowywane były dynamiczne widoki systemowe. Ponadto w katalogu głównym umieściłem plik dk209472_pracamgr.pdf, z którego wydrukowałem tę pracę.

A.2. Instalacja

System LoXiM został napisany pod system operacyjny Linux. W celu jego zainstalowania należy:

- przegrać katalog szbd na dysk twardy
- skompilować go poleceniem make
- uruchomić serwer poleceniem ./Server znajdującym się w katalog szbd/LoximServer
- uruchomić klienta poleceniem ./Lsbql w katalogu szbd/LoximClient
- wpisać login i hasło (np. root/root)

Bibliografia

[Lon08] Kevin Loney, Bob Bryla, *Oracle Database 10g*, Helion, 2008.

[Gur02] Mark Gurry, *Optymalizacja Oracle SQL. Leksykon kieszonkowy*, Helion, 2002.

[Wry02] Stanisław Wrycza, Bartosz Marcinkowski, Krzysztof Wyrzykowski, *Język UML 2.0 w modelowaniu systemów informatycznych*, Helion, 2006.

[Eck02] Bruce Eckel, *Thinking in C++. Edycja polska*, Helion, 2002.

[Sub04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, PJWSTK, Warszawa 2004.