МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних систем та мереж

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
комп'ютернихсистем та мереж

_____Жуков І.А.

«___» _____202_ р.

ДИПЛОМНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬО–КВАЛІФІКАЦІЙНОГО РІВНЯ
"МАГІСТР"
напрям підготовки - 123 "Комп'ютерна інженерія"

Тема: Методи організації високоефективної обробки даних в *Oracle* системах.

Виконавець: Яковенко О.Ф

Керівник: Лукашенко В.В.

Нормоконтролер: Андрєєв В.І.

Засвідчую, що у магістерській роботі немає запозичень праць інших авторів без відповідних посилань

Студент _____ Яковенко О.Ф.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних систем та мереж
Освітнього ступеня магістр
Напрям 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____Жуков І.А.

«___» _____2020 р.

ЗАВДАННЯ
на виконання дипломного проекту
Яковенко Олександра Федоровича
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема проекту: " Методи організації високоефективної обробки даних в *Oracle* системах ".
затверджена наказом ректора від "25" вересня 2020 року № 1793/ст.

2. Термін виконання проекту (роботи): з 25.09.2020 до 21.12.2020

3. Вихідні дані до проекту (роботи): Інформація про ефективне використання апаратних засобів для налаштування бази даних *Oracle*. Сучасні методи налаштувань, які використовуються для проектування ефективної схеми *Oracle*. Рекомендації програмування на мові програмування на *PL/SQL*

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):
1) аналіз архітектурних рішень для налаштування продуктивності;
2) дослідження паралелізму, багатоверсійності та структури пам'яті;
3) проектування ефективної схеми, рекомендації програмування на *PL/SQL*;

5. Перелік обов'язкового графічного матеріалу:
1) Презентація *PowerPoint*;

6. Календарний план-графік

№ п/п	Етапи виконання дипломного проекту	Термін виконання етапів	Примітка
1	Ознайомитись з постановкою задачі дипломного проектування	05.10.2020р. – 12.10.2020р.	
2	Вивчити спеціальну літературу і технічну документацію	13.10.2020р. – 17.10.2020р.	
3	Проаналізувати архітектурні рішення для налаштування продуктивності в *Oracle*	18.10.2020р. – 20.10.2020р.	
4	Написати розділ дипломного проекту щодо аналізу архітектурних рішень для налаштування продуктивності	21.10.2020р. – 28.10.2020р.	
5	Проаналізувати структури пам’яті *Oracle*	29. 10.2020р. – 2.11.2020р.	
6	Написати розділ дипломного проекту щодо дослідження структур пам’яті, паралелізму та узгодженого читання	3.11.2020р. – 13.11.2020р.	
7	Проаналізувати проектування ефективної схеми	14. 11.2020р. – 21.11.2020р.	
8	Написати розділ дипломного проекту з описом етапів проектування ефективної схеми та рекомендацій програмування на мові *PL/SQL*	22. 11.2020р. – 01.12.2020р.	
9	Підготовити графічний демонстраційний матеріал	02. 12.2020р. – 09.12.2020р.	

7. Дата видачі завдання «26» вересня 2020 р.

Керівник дипломного проекту Лукашенко В.В.
(підпис)

Завдання прийняв до виконання Яковенко О.Ф.
(підпис випускника) (П.І.Б.)

Дата ___.___._____

РЕФЕРАТ

Пояснювальна записка до дипломного проекту “Методи організації високоефективної обробки даних в *Oracle* системах”: 87 с., 17 рис., 2 табл., 20 літературних джерел.

ТЕХНОЛОГІЇ *ORACLE*, ВИДІЛЕНИЙ СЕРВЕР, РОЗПОДІЛЕНИЙ СЕРВЕР, ТРАНЗАКЦІЇ, *SQL*, *PL / SQL*, БАЗИ ДАНИХ, СХЕМА, КЕШ, РІВНІ ІЗОЛЯЦІЇ.

Мета дипломного проекту – дослідити методи, які впливають на зберігання та обробку даних в *Oracle* системах.

Об’єкт дослідження – реляційна база даних *Oracle.*

Предмет дослідження – методи, які пов'язані з ефективною обробкою даних в реляційних системах *Oracle*.

Метод дослідження – аналіз сучасних сучасних архітектурних рішень що впливають на продуктивність бази даних, дослідження налаштувань для проектування високоефективної схеми *Oracle*, використання оперативної пам’яті та теоретичне дослідження на масштабованіть.

Новизна – Найбільш суттєвими науковими результатами дипломного проекту є порівняння існуючих методів проектування ефективних схем *Oracle*. Обгрунтування переваг та недоліків даних методів, які включають як вибір апаратної частини так і програмних налаштувань. Сформування рекомендацій щодо ефективного використання даних методів при проектуванні схем та щодо будування високоефективних запитів на мові програмування *PL/SQL*.

Актуальність – тема вибору правильних архітектурних рішень при проектуванні ефективної схеми є дуже важливою для будь-якої організації, яка вирішить зберігати важливу для неї інформацію в реляційних базах даних.

В роботі досліджуються питання технічного і економічного характеру, які актуальні сьогодні на ринку і в подальшому впливають на продуктивність та масштабованіть бази даних.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

БТС – багатопоточний сервер.

ОС – операційна система.

API – прикладний програмний інтерфейс.

IDE – інтегроване середовище розробки.

I/O – ввід/вивід.

GUI – графічний інтерфейс користувача.

PGA – глобальна область процесу.

UGA – глобальна область користувача.

SGA – системна глобальна область.

ВСТУП

Дана робота буде корисна для розробників, яким необхідні більш сучасчі методи проектування і будування масштабованих систем з використанням платформи *Oracle*. Її буде важко читати людям, які тільки починають вивчати мову *SQL* та технології, як *JDBC і ODBC.* Вона розрахована на розробників, які знають, вміють писати базові запити до бази дани, та вводити їх завдяки *SQL Plus* і т.д. Ця робота не навчить вас *SQL* – вона розровість вам про те, що необхідно знати для створення «грамотних» *SQL*-запитів. Вона також не навчить, як створювати код додатків, але розповість про те, що необхідно знати для написання «грамотних» додатків, заснованих на *Oracle*.

Я намагатимусь розповісти про найбільш важливі, на мою думку, питання, а саме – база даних *Oracle* та її архітектура. Можно було б аналогічним чином розповісти про розробку додатків з використанням конкретної технології обміну даними з *Oracle,* наприклад *ODBC*, яку використовують програмісти на *Visual Basic,* або *JDBC* і *EJB* які використовують прорамісти на *Java*. Та в цій роботі не віддається перевага якій-небудь конкретній архітектурі додатків. Замість цього і ній описані можливості бази даних і пояснені особливості її роботи. Оскільки база даних – ядро архітектури любого додатку, робота повинная зацікавити широкий круг читачів, в особистості тих, які працюють з клієнт серверною архітектурою.

За основу було взято найпопулярніші версії *Oracle*: *Oracle9i, Oracle10g*. Зараз вони є оптимальним варіантом як в технічному так і в економічному плані, якщо вибирати з поміж усіх баз даних.

Особли увагу було приведено рогзляду фізичних структур баз даних таких як таблиці, індекси і типи даних, висвітлюючи при цьому технології іх оптимального використання.

РОЗДІЛ 1
АНАЛІЗ АРХІТЕКТУРНИХ РІШЕНЬ ДЛЯ НАЛАШТУВАННЯ ПРОДУКТИВНОСТІ

У цьому розділі обговорюються деякі з архітектурних (на макрорівні) рішень, які можуть допомогти (або перешкодити) в досягненні можливої для Oracle масштабованості та продуктивності. Будуть розглянуті дві головні цілі:

- **Досягнення високого рівня паралельної роботи** Порівняння організації з'єднання і кластеризації на виділеному сервері і на спільно використовуваному сервері.
- **Ефективне управління простором пам'яті** Локально керовані табличні простору і секціонування (*partitioning*).

Я не буду заглиблюватися в деталі архітектури процесів сервера *Oracle* і структур пам'яті або ж пояснювати, яким чином можна налаштувати секціонування або паралельну обробку, а покажу, як і коли можна використовувати ці засоби *Oracle*.

1.1.Порівняння режимів виділеного і розділеного сервера

Найбільш загальним методом з'єднання з базою даних *Oracle* є, безумовно, використання виділеного сервера. Його легко сконфігурувати, і він спрощує процеси налагодження і трасування. Конфігурація спільно використовуваного сервера складніша, але вона може виявитися незамінною, якщо потрібно підключити більшу кількість сесій при використанні того ж самого апаратного забезпечення (кількість процесорів і пам'яті).

Конфігурація розділеного (спільно використовуваного) сервера в ранніх випусках *Oracle* відома як багатопоточний сервер (БТС). БТС і розділений сервер - це синоніми. У цій роботі застосовується термін «Розділений сервер».

Коротко розглянемо, як встановлюються з'єднання в конфігурації спільно використовуваного сервера і виділеного сервера, і порівняємо ці методи.

«Чи слід працювати з конфігурацією спільно використовуваного сервера Oracle?»

Не варто застосовувати спільно використовуваний сервер, поки одне тимчасових з'єднань до бази даних не буде більше, ніж може обробити операційна система. До тих пір поки машина не перегружена процесами / потоками, слід використовувати виділений сервер. Коли ж кількість звернень до машини досягне свого максимального порога і чергове з'єднання буде або неможливо, або має негативного впливу на продуктивність, тоді має сенс перейти на спільно використовуваний сервер. На щастя, клієнт не може цим керувати, так що для програми не складе труднощів переключитися з однієї конфігурації на іншу.

Існує цілий ряд причин, які обумовлюють вибір виділеного сервера, але однією з головних є те, що якщо виділений сервер не перевантажений процесами/ потоками, робота в такій системі буде проводитися швидко.

1.1.1. Як працює з'єднання з виділеним сервером

Фактично при з'єднанні з екземпляром *Oracle* здійснюється з'єднання з безліччю процесів і із загальною пам'яттю. База даних *Oracle* є складним набором взаємодіючих процесів, які всі разом працюють над вирішенням поставленого завдання.

При з'єднанні з виділеним сервером по мережі виконуються наступні дії:

1. Клієнтський процес з'єднується по мережі з приймачем, який зазвичай запущений на сервері бази даних. Однак бувають ситуації, коли приймачі і екземпляр сервера бази даних пріцюють на різних машинах.

2. Приймач створює новий виділений процес (*Unix*) або просить базу даних створити новий виділений потік (*Windows*) для цього з’єднання. Що буде

створено процес або потік, визначають розробником ядра *Oracle* в залежності від операційної системи. Вибирається реалізація, що підходить для даної операційної системи.

3. Приймач в автоматичному режимі підключається до новостворенного процесу або потоку.

4. Клієнтський процес посилає запити до виділеного сервера, той їх обробляє і відправляє результати назад.

На рис. 1.1. показано, яким чином працює конфігурація виділеного сервера. На одних платформах (наприклад, в *Windows*) екземпляр *Oracle* буде єдиним процесом з потоками, а на інших (наприклад, в *Unix*) кожна «бульбашка» буде окремим фізичним процесом. У конфігурації виділеного сервера у кожного клієнта є його особистий, пов'язаний з ним процес / потік.

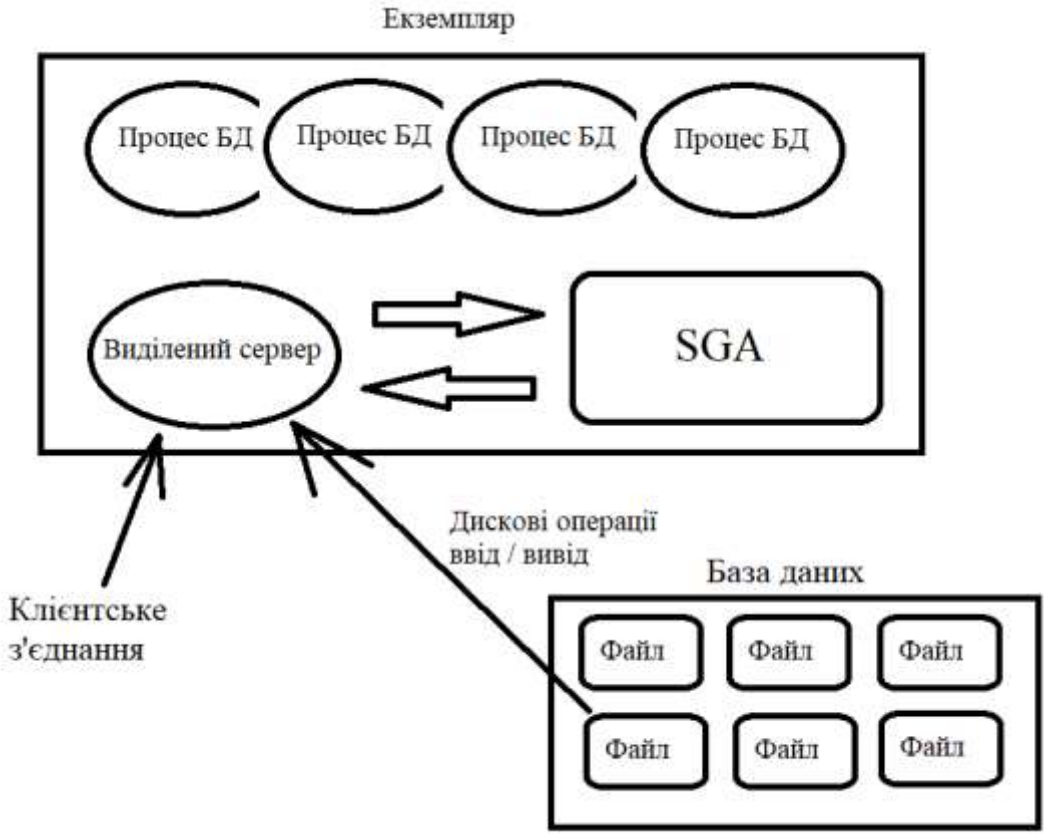

Рис. 1.1. Процес з’єднання в конфігурації виділеного сервера

Виділений сервер буде отримувати і виконувати *SQL* користувача. Він буде читати файли даних і розміщувати дані в кеші, продивлятися кеш бази в

пошуках даних, виконувати оператори *UPDATE* і запускати код *PL / SQL*. Його основною метою є відповіді на звернені до нього виклики *SQL*.

Таким чином, в режимі виділеного сервера існує взаємно однозначна відповідність між процесами в базі даних і клієнтським процесом. У кожного з'єднання з виділеним сервером є свій, виключно для нього призначений виділений сервер. Це не означає, що у кожної окремої сесії є свій виділений сервер, його наявність можливо для програми, що використовує окреме фізичне з'єднання (виділений сервер) для паралельної роботи багатьох сесій. Однако в загальному випадку в режимі виділеного сервера існує взаємно однозначне відношення між сесією і виділеним сервером.

1.1.2. За і проти використання з'єднання з виділеним сервером

Нижче перераховані переваги використання з'єднання виділеного сервера з базою даних:

- Легко встановлюється. Насправді, не потрібно майже ніякої установки.
- Є найбільш швидким режимом роботи з базою даних, оскільки вимагає найменшого обсягу кодування.
- Оскільки файли трасування пов'язані з процесом (наприклад, з виділеним сервером), такі засоби, як *SQL_TRACE*, вигідно використовувати в режимі виділеного сервера. Насправді, спроби використовувати *SQL_TRACE* при з'єднанні не в режимі виділеного сервера можуть виявитися марними.
- Доступні всі адміністративні функції. (Наприклад, в режимі спільно використовуваного сервера недоступні певні адміністративні засоби, такі як запуск і завершення роботи бази даних і можливості відновлення.)
- Пам'ять для *UGA* (призначена для користувача глобальна область), є пам'яттю, яка визначається сесією, динамічно розміщується в області глобальних процесів (*PGA*) і не потребує в конфігоруванні в загальній глобальної області (*SGA*).

Але існують і деякі недоліки використання з'єднання виділеного сервера:

- Процеси / потоки клієнта поглинають в процесі роботи сесії ресурси сервера (пам'ять, цикли процесора для перемикання контексту і т.д.).
- Збільшення кількості користувачів може привести до перегрузки операційної системи великою кількістю процесів / потоків.
- Обмежується управління кількістю одночасно активних сесій, хоча в *Oracle9i* можна звернутися до менеджера ресурсів (*Resource Manager*) і встановити максимальну кількість активних сесій в правилі для груп.
- Неможливо використовувати цей режим з'єднання для концентрації цій каналів зв'язку бази даних.
- Непродуктивні витрати від процесу, створення і знищення потоку можуть бути дуже великими при високому рівні фактичних з'єднань / відключень. В такому випадку кращим буде з'єднання з спільно використовуваних сервером. Зазначу, що для більшості *web*-додатків це не є проблемою, оскільки зазвичай вони самостійно виконують деякі види з'єднань.

Сьогодні на більшості звичайних серверів баз даних (як правило, машини з кількістю процесорів від двох до чотирьох) можуть працювати від 200 до 400 одночасних підключень. На великих машинах ця межа значно вище.

Не існує чітких обмежень для використання конфігурації виділеного сервера. У ряді випадків доводиться відмовлятися від застосування цієї конфігурації вже при підключенні сотні користувачів. Однак можна зустріти і такі системи, що використовують конфігурацію виділеного сервера, які спокійно працюють при підключенні понад 1000 користувачів. Перехід до спільно використовуваного сервер визначається можливостями апаратного забезпечення і тим, як користуються базою даних. До тих пір поки процесор і оперативна пам’ять справляються з навантаженням, виділений сервер є найпростішою і найбільш функціональної конфігурацією.

1.1.3. Як працює з'єднання з спільно використовуваних сервером

Протоколи з’єднання з спільно використовуваним сервером і з виділеним сервером дуже відрізняються. У конфігурації спільно використовуваного сервера не існує взаємооднозначної відповідності між клієнтами (сесіями) і серверними

процесами / потоками. Застосовується пул іменованих загальних серверів, які виробляють ті ж операції, що і виділений сервер, але для безлічі сесій, замість однієї.

Крім того, при застосуванні з'єднання з спільно використовуваним сервером працюють процеси, відомі як *диспетчери*. Клієнт з'єднується з диспетчером, який організовує передачу запиту клієнта до спільно використовуваного сервера і повернення відповіді клієнту після завершення роботи сервера. Ці диспетчери запускаються в базі даних процесом прийому інформації. При встановленні з'єднання приймач інформації підключає користувача до доступного диспетчеру.

При підключенні за допомогою бездротової технології з спільно використовуваним сервером зазвичай відбувається наступне:

1. Процес клієнта з'єднується по мережі з запущеним на сервері бази даних приймачем інформації, який вибирає для подальшого підключення диспетчера з пулу доступних діспетчерів.

2. Приймач інформації посилає клієнту адресу обраного для з'єднання диспетчера або безпосередньо перенаправляє з'єднання до диспетчеру (в деяких випадках з'єднання може бути переведено напряму). Це відбувається на рівні *TCP / IP* з'єднання і залежить від операційної системи. Якщо ж безпосереднє перенаправлення до диспетчера не виконується, клієнт відключається від приймача інформації та підключається до диспетчеру.

3. Процес клієнта відправляє запит цьому диспетчеру.

На рис. 1.2. показаний процес підключення в режимі спільно використовуваного сервера.

Отримавши запит з'єднання, приймач інформації вибирає процес диспетчера з області доступних диспетчерів. Потім він відправляє клієнту інформацію про з'єднання, що описує, яким чином клієнт може підключитися до процесу диспетчера. Це повинно бути зроблено, тому що приймач інформації працює на хості з відомим ім'ям і портом, а диспетчери приймають з'єднання на випадково призначених портах на цьому сервері. Приймач інформації знає про ці випадково назначені порти і вибирає диспетчера для клієнта. Потім клієнт

відключається від приймача інформації та безпосередньо з'єднується з диспетчером. І в результаті фізичне з'єднання з базою даних установлено.

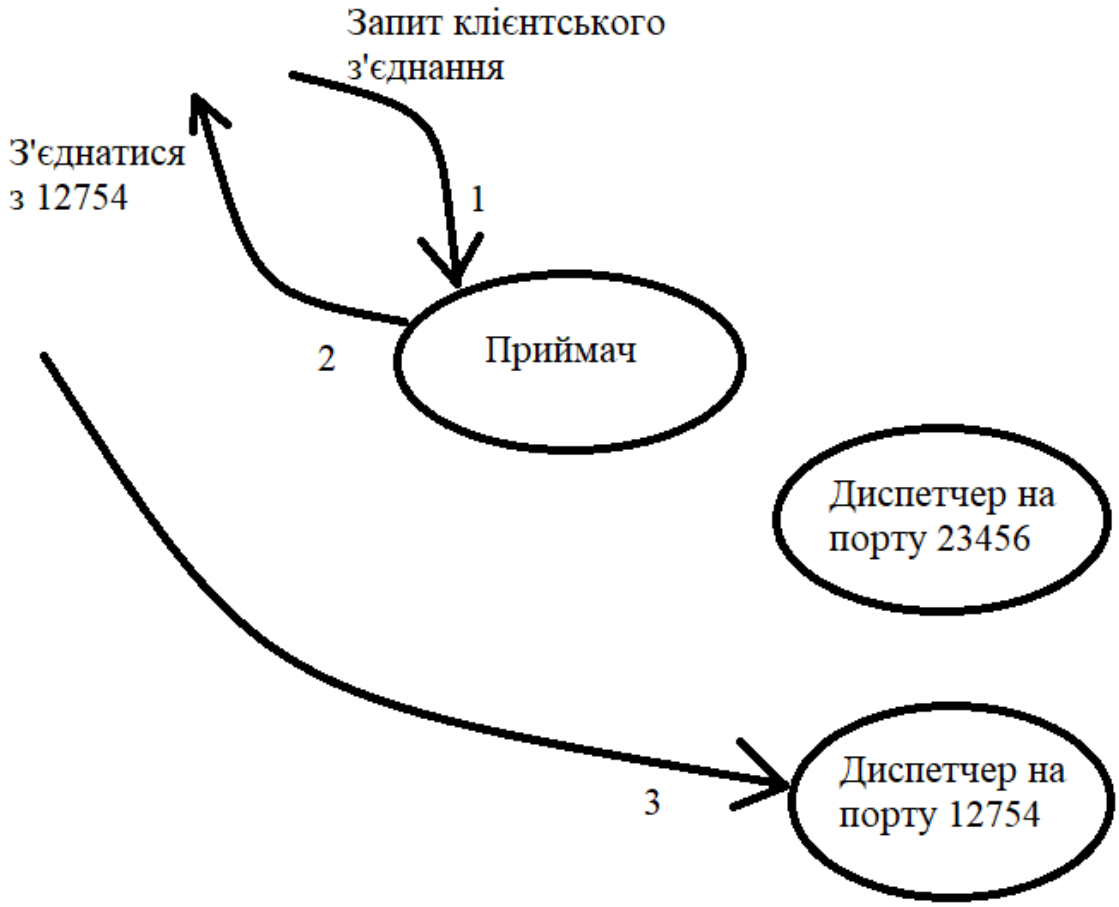

Рис. 1.2. Процес підключення в режимі спільно використовуваного сервера

1.1.4. Обробка команд спільно використовуваного сервера

Наступним кроком після встановлення з'єднання є обробка запитів. У разі виділеного сервера достатньо відправити напряму запит (*select * from emp*), наприклад для обробки. У разі спільно вікористовуваного сервера це зробити набагато складніше. Більше немає окремого виділеного сервера. Доведеться шукати інший спосіб звернення до процесу, який зможе обробити запит. Для цього робляться такі дії:

1. Диспетчер поміщає запит в загальну (доступну всім диспетчерам) чергу в *SGA*.

2. Перший доступний спільно використовуваний сервер забирає і про-розробляє цей запит.

3. Спільно переглянуте сервер поміщає відповідь на запит в чергу відповідей, незалежну від черг, які були організовані диспетчером в *SGA*.

4. Диспетчер, до якого був підключений клієнт, забирає цю відповідь і відсилає його клієнту.

Цей процес показаний на рис. 1.3.

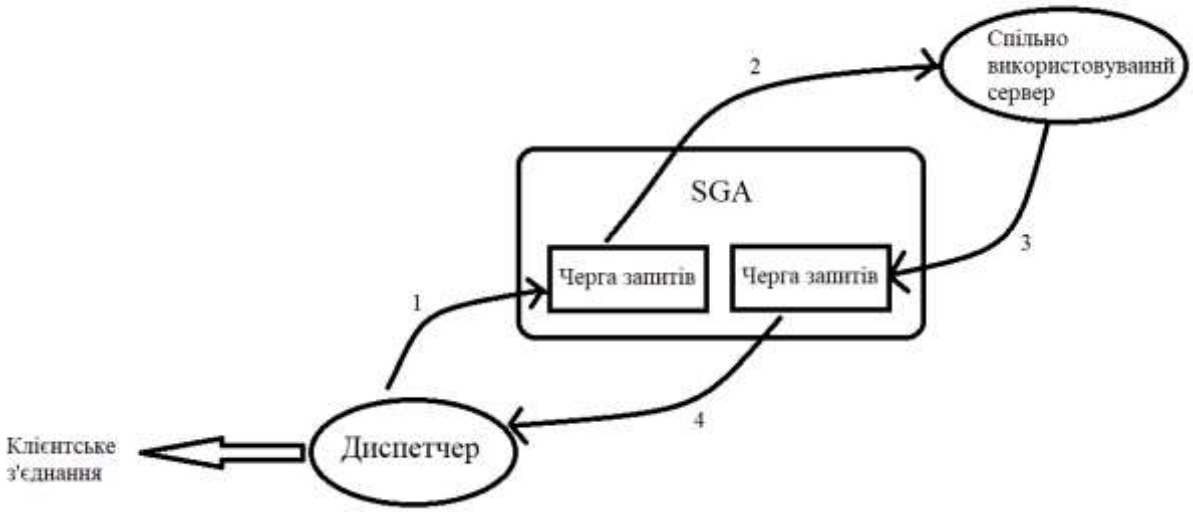

Рис. 1.3. Обробка запитів в конфігурації спільно використовуваного сервера

Цікаво відмітити, що ці дії виконуються для кожного виклику бази даних. Відповідно, вимоги на аналіз запиту може оброблятися спільно використовуваних сервером 1, вимога на вибірку першого рядка в запиті може виконуватися спільно використовуваним сервером 2, наступного рядка - сервером 3 і, нарешті, спільно використовуваний сервер 4 може закривати набір результатів.

Видно, що шлях коду від передачі запиту (деякого *SQL*) клієнтом до отримання відповіді є довгим і звивистим. Однак ця архітектура добре працює при необхідності масштабування системи. У разі з'єднання з виділеним сервером кожна сесія - це новий процес. В випадку з'єднання з спільно використовуваним сервером кількість процесів регульовано і фіксоване (або, щонайменше, установлюється в межах діапазону швидкої роботи апаратного забезпечення). Для того щоб отримати з'єднання $N + 1$, необхідно достатній обсяг доступної

пам'яті в *SGA* для цієї сесії *UGA* (пам'ять використовується сесією для змінних стану, курсорів та інших об'єк тів).

1.1.5. За і проти з'єднання з спільно використовуваних сервером

Нижче представлені деякі переваги від застосування сполуки спільно використовуваного сервера з базою даних:

- Це з'єднання робить систему більш масштабованою. На машині, яка здатна фізично обробити сотню з'єднань виділеного сервера, можна, в залежності від програми, обробити більше 1000 з'єднань спільно використовуваного сервера. Якщо застосовується традиційний тип програми, клієнт / сервер, коли більшість сесій з'єднань інертні (наприклад, з 1000 з'єднань, може бути, від 50 до 100 що працюють одночасно але, а решта чекають користувача, що знаходиться в роздумах), то конфігурація спільно використовуваного сервера може виявитися досить корисною для масштабованості додатку. Відзначимо, що це масштабування стосується тільки процесора, оскільки для цих 1000 з'єднань необхідна достатня кількість пам'яти.
- Це з'єднання дозволяє проводити високоточний контроль над можливою кількістю одночасно працюючих сесій і устанавлює жорстке обмеження на загальну кількість активних сесій.
- Використовуються менше пам'яті, ніж в конфігурації виділеного сервера.
- Можна зареєструвати $N + 1$ користувача, на що конфігурація виділеного сервера не здатна.
- При високому рівні з'єднань / відключень робота в конфігурації спільно використовуваного сервера може виконуватися швидше, ніж в конфігурації виділеного сервера. Процес створення і знищення процесів і потоків є дуже дорогим і може перевищити накладні витрати від довгого шляху коду.

До недоліків конфігурації спільно використовуваного сервера відноситься наступне:

- Ця конфігурація складніше в установці, ніж конфігурація виділеного сервера, хоча кошти *GUI* (графічний інтерфейс користувача), які поставляються разом з базою данних, декілька згладжують ці трудності. Більшість проблем установки з тим, що багато хто намагається вручну редагувати конфігурацію користувача), що поставляються разом з базою даних, замість того, щоб користуватися спеціальними інструментами.

- У тій системі, в якій добре працює з'єднання з виділеним сервером, з'єднання з спільно використовуваним сервером буде працювати повільніше. Шлях коду спільно використовуваного сервера, за визначенням, довше, ніж шлях з'єднання з спільно використовуваним сервером.

- У конфігурації спільно використовуваного сервера стають недоступними деякі можливості бази даних. Наприклад, неможливо запустити або завершити роботу екземпляру *Oracle*, виконати відновлення носія і використовувати *Log Miner*. Для роботи з цими засобами необхідно з'єднуватися з базою даних в режимі виділеного сервера.

- Пам'ять, необхідна для всіх одночасно підключених сесій *UGA*, підраховується і налаштована як частина *LARGE_POOL* файлу настройки init.ora. Ви повинні розуміти, що максимальний рівень з'єднань і розмір памяті відповідають один одному. У *Oracle9i* випуску 2 є можливість динамічного перевизначення налаштувань *LARGE_POOL*, так що можна управляти використанням пам'яті і змінювати її (але в ручному, а не в автоматичному режимі). При роботі спільно використовуваного сервера трасування (застосування *SQL_TRACE*) стає неможливою. Файли трасування точно вказують на процес, а в режимі спільно використовуваного сервера процеси не вказують на сесію. Таким чином, оператори *SQL* конкретної сесії будуть записані в різні файли трасування, і оператори *SQL* інших сесій будуть записані в ті ж самі файли. Одним словом, неможливо визначити де чиї оператори.

- Існує ймовірність штучних блокувань. Якщо сумісно використовуваний сервер запустив обробку запиту, то повернення не буде до тих пір, поки не буде завершено обробку цього запиту. Так, якщо спільно використовуваному серверу був відправлений запит *UPDATE* і рядок вже заблокована, то і сервер буде

заблокований. Це штучна ситуація блокування, при якій блокуючий засіб не може звільнити заблокованих до тих пір, поки у заблокованих обмежений доступ до всіх спільно використовуваних серверів.

- Існує ймовірність монополізації спільно використовуваного сервера. Це може статися з тієї ж самої причини, що й штучне блокування. При запуску коли довго виконуються транзакції сесія монополізує загальний ресурс, перешкоджаючи роботі інших. Якщо ж таких тривалих транзакцій занадто багато, робота системи стає нескінченною, тому що необхідно чекати їх завершення.

Таким чином, в універсальних випадках не варто застосовувати конфігурацію спільно використовуваного сервера. Про нього слід згадати, тільки якщо база даних фізично не справляється з навантаженням.

1.2.Переваги кластеризації

Для підвищення працездатності і забезпечення горизонтальної масштабованості не існує нічого кращого, ніж кластеризація.

Кластеризація - це конфігурація апаратного забезпечення, де безліч комп'ютерів фізично з'єднуються разом для виконання одного завдання на єдиному наборі дисків. Кластеризація - добре описана технологія, випробована компанією *Equipment Corporation (DEC)* на машинах *VAX / VMS* в 1980-их рр. Сьогодні використання кластерів простягається від професійних потужних машин до груп портативних комп'ютерів, що працюють під управлінням *Linux*. У машин в кластері є своя мережа для взаємодії, відома під назвою «быстродействующая внутренная связь». Її реализація може варіюватися в залежності від продавця: від приватних рішень високого класу до мережі, побудованої на *Linux*. Ця мережа забезпечує високу швидкість взаємодії машин в кластері.

У минулому кластеризація була технологією, доступною для обраних. Була потрібна спеціальна конфігурація апаратного забезпечення, зазвичай підтримувана постачальниками тільки на високому рівні. Вартість такого

обладнання була досить високою. Тепер продукція зі здатністю до кластеризації - це готові машини (які поставляються *Dell* і *Hewlett-Packard*) з такими операційними системами, як *Linux*. За ціну від $ 10 000 до $ 20 000 можна отримати повнофункціональне апаратне забезпечення, яке працює цілодобово з можливістю кластеризації від 4 до 8 процесорів. Все це потребує наявності двох серверів кабелю.

Горизонтальне масштабування - це здатність додавання системної потужності за рахунок збільшення кількості машин. У минулому використовувалось вертикальне масштабування, здійснюване за допомогою збільшення кількості процесорів, збільшення оперативної пам'яті і т.п. на одній машині. Вертикальне масштабування добре працює до тих пір, поки не вичерпуються можливості машини. Наприклад, звичайна машина з *Unix* не здатна масштабуватися з 64 до 128 процесорів. Горизонтальне масштабування прибирає всі ці бар'єри. Кожна машина може мати жорсткі обмеження, але декілька машин в кластері їх спокійно подолають. Горизонтальне масштабування забезпечує можливості додавання комп'ютерних ресурсів в міру необхідності.

Висока доступність - це одна з видатних можливостей кластерів. Кластери, за визначенням, є високо доступними платформами. Поки хоча б один комп'ютер в кластері продовжує роботу, ресурси цього кластера є доступними. Відмова одного з компьютерів не є перешкодою доступу до бази даних.

У цій главі розглядається конфігурація *Oracle*, так звана *Real Application Clusters (RAC)*. Також розглянемо, як *Oracle* працює на кластерах, і досліджуємо, яким чином *RAC* забезпечує горизонтальну масштабованість і високу доступність.

Давайте уявимо що ми створюємо велику базу даних з великою кількістю користувачів. Протягом першого року вона буде працювати з сотнею користувачів, в наступному їх стане 500 осіб, і очікується подальшое зростання числа користувачів. Але в цьому році потреби вельми скромні. Яким чином нам краще вчинити?

В даному випадку можна застосувати один з наступних методів:

- Купити найбільшу машину, яку тільки можливо. Це забезпечить необхідну потужність на найближчі два роки. Єдина проблема полягає в тому, що зараз витратити велику кількість грошей на те, що через два роки буде коштувати в 10 разів менше.
- Купити машину, яка відповідає сьогоднішнім вимогам. Потім купити таку ж машину через рік, але вже за половину вартості і в два рази швидше, а потім купити ще одну і т.д. Одним словом, ви купуєте тільки те, що необхідно сьогодні. Єдине застереження потрібно переконатися в тому, що апаратне забезпечення, яке купується здатне до кластеризації. Це питання до виробника електронного пристрою.

1.2.1. Принцип роботи *RAC*

База даних може бути встановлена і відкрита одночасно багатьма екземплярами, і це як раз те, чим займається *RAC*. У чому різниця між екземпляром і базою даних?

Екземпляр - це безліч процесів і пам'ять, яка використовується цими про-процесами (*SGA*). Екземпляр *Oracle* взагалі не потрібно пов'язувати з базою даних. Екземпляр *Oracle* можна запустити без якої б то не було бази даних. При роботі в цьому режимі екземпляр не дуже корисний, але це цілком можливо.

База даних - це безліч файлів, що містять дані: журнал бази даних, елементи управління, дані і тимчасові файли. У бази даних немає процесів і пам'яті, вона містить лише файли.

1.2.2. Екзмепляри в *RAC*

У середовищі *RAC* екземпляр *Oracle* запускається на кожній машині (також відомої, як вузол) в кластері. Кожен з цих екземплярів встановлює і відкриває одну і ту ж базу даних, так як кластери спільно використовують диски. Важливо мати на увазі, що робота здійснюється тільки з однією базою даних. У кожного екземпляра *Oracle* есть повний доступ на читання / запис до кожного байту в базі даних.

Кожен екземпляр вважається рівним будь-якому іншому екземпляру в кластері. Можна виконати оновлення таблиці *EMP* з вузла 1. Можна виконати оновлення таблиці *EMP* з вузла 2. Насправді, можна оновити запис *where ename='KING'* з обох вузлів - і 1, і 2, слідуючи тим же правилам, як якщо б використовувався єдиний екземпляр *Oracle*. Механізми блокування та управління одночасним доступом, які застосовуються при роботі в єдиному екземплярі *Oracle* або єдиної бази даних, діють і при запуску *RAC* для *Oracle*.

На рис. 1.4. показаний *RAC* при роботі з кластером з чотирьох вузлів (чотирьох комп'ютерів). Кожен вузол буде запускати свій власний екземпляр *Oracle* (процеси / пам'ять). Існує єдина копія бази даних, спільно використовувана всіма вузлами в кластері. У кожного з них є повний доступ на читання / запис до кожного байту даних в базі даних.

В основі *Oracle RAC* лежить те, що запускається багато екземплярів *Oracle*. Кожний екземпляр існує автономно, рівноправно. Немає ніякого головного екземпляру, так само як немає єдиної точки відмови. Кожен екземпляр буде встановлювати і відкривати одну і ту ж базу даних, одні і ті ж файли. Екземпляри будуть працювати разом, зберігаючи узгодженість роботи кеша, уникаючи переписування змін, зроблених іншими екземплярами, і читання застарілої інформації. Якщо один з них з будь-якої причини зазнає невдачі (через відмову апаратного або програмного забезпечення), будь-який інший з решти екземплярів в кластері відновить роботу, яку виконував відмовивший екземпляр, і продовжить операції.

Фізично існує одна база даних з безліччю екземплярів. Для клієнтських додатків існує тільки одна служба, з якою вони з'єднуються, і ця служба підключає їх до екземпляру, який забезпечує повний доступ читання / запису до бази даних. Про те, як це відбувається, додаток не знає. Будь-який додаток, який працює з *Oracle9i*, буде точно так же працювати в кластері *Oracie9i RAC*.

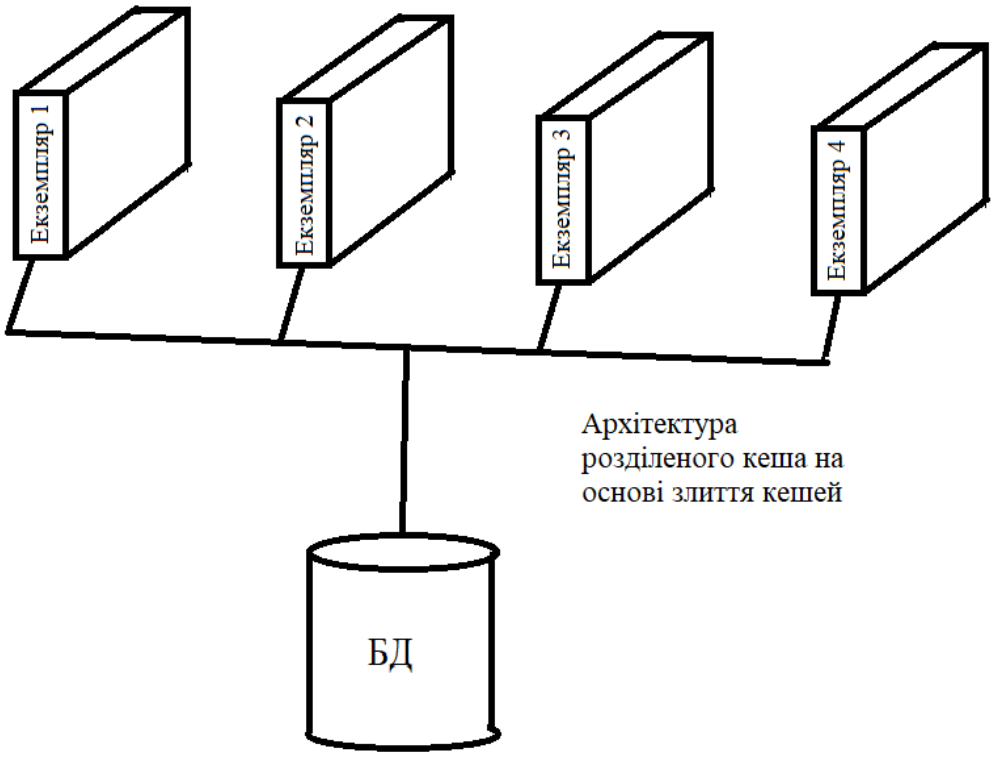

Рис. 1.4. *RAC* з кластером із чотирьох вузлів

RAC не вирішує всіх проблем, пов'язаних з продуктивністю. В екстремальних випадках додатки будуть працювати повільніше. Наприклад, якщо є система, що виконує надмірний розбір (не використовуючи змінних прив'язки), то ви виявите, що, оскільки операції бібліотечного кешу глобально координуються в кластері, проблема, що була у єдиного екземпляру, в кластері багаторазово зросте. Кластеризація в багатьох випадках тільки посилює проблеми поганої продуктивності погано спроектованої системи. За допомогою *RAC* неможливо виключити такі складності, як:

- *Конфлікт бібліотечного кешу*. Якщо є проблеми з бібліотечним кешем в єдиному екземплярі, то безліч екземпляров тільки посилить.
- *Надмірний ввід / вивід*. Так як ввід / вивід координується глобально, проблеми введення / виведення, що мають місце в конфігурації єдиного екземпляру, будуть зростати при використанні конфігурації безлічі екземплярів.
- *Блокування*. Проблем з блокуванням особливо багато в системі, яка застосовує в таблицях лічильники замість послідовностей. Ця реалізація

викликає конфлікт в єдиному екземплярі, і цей конфлікт багаторазово збільшиться в кластерізованний середовищі.

Ці проектні недоробки повинні бути виправлені за допомогою методів реалізації. Наприклад, конфлікт бібліотечного кеша може бути зменшений за рахунок відповідного використання змінних прив'язки і більш частого розбору операторів *SQL*. Можна скоротити надмірний ввід / вивід шляхом реалізації фізичних структур, які найбільше підходять для способу використання даних.

Можна видалити або скоротити проблеми з блокуванням за допомогою вбудованих засобів бази даних. таких як послідовності. *RAC Oracle9i* допоможе в масштабуванні, але не виправить систему яка уже відмовила!

1.2.3. Переваги використання *RAC*

Було з'ясовано, як працює *RAC*. Теперь коротоко розглянемо, що таке *RAC* для бази даних і що таке *RAID* для дисків. Якщо диск в *RAID*-масиві вийде з ладу, не відбудеться нічого страшного - сьогодні в більшості систем можна зробити його «гарячу заміну». Якщо один з екземплярів в *RAC* зазнає невдачі, теж нічого страшного інші екземпляри автоматично замінять його. Помилковий екземпляр можна буде виправити, усунувши проблему з програмним або апаратним забезпеченням яка викликала його відмову.

RAC є рішенням, що гарантує, що після того, як база даних буде запущена і почне роботу, ніякі катастрофи не зможуть нанести їй шкоду. Так що можна захистити диски за допомогою *RAID* або інших технологій, мережі за допомогою додаткового обладнання, а базу даних за допомогою резервних екземплярів *Oracle*.

Сьогодні наявність у машини чотирьох, вісьмох і навіть більшої кількості процесорів вважається нормальним. Те ж саме відбувається зараз з технологією кластеризації. Рівень її використання поступово зростає. Для підтримки кластера більше не потрібно купувати потужне і дороге обладнання. Ця технологія є зрілою. Кластеризація перестає бути чимось неординарним.

Доречі, концепція роботи з *RAC* добре документована *Oracle*. Зокрема, один з посібників називається *Real Applications Clusters Concepts Guide* (так,

інший *Concepts Guide*). У ньому можна знайти інформацію про фізичну реалізацію *RAC*, про організацію узгодженої роботи безлічі кешей (засіб *cache fusion* (злиття кешей) і т.п. В *RAC Concepts Guide* є розділ, присвячений тому, що неможливо масштабувати за допомогою *RAC* і як бути в цьому випадку. Наприклад, там описано, яким чином паралельний запит може бути оброблений не тільки всіма доступними процесорами єдиної машини, але і всіма екземплярами в *RAC*-кластері, запит може вивиконуватися паралельно на кожному з *N* вузлів в кластері.

1.3. Застосування секціонування

Секціонування - це здатність бази даних фізично розбивати дуже великі таблиці або індекси на менші, більш керовані частини. Так само, як буває корисно розбити паралельні процеси на менші частини для незалежної обробки, секціонування може бути використано для розбивки дуже великих таблиць / індексів на декілька маленьких, незалежно керованих шматочків.

Необхідно чітко розуміти фізичну реалізацію секціонування. Те, яким чином секціонування буде застосовуватися, залежить від поставленої кінцевої мети.

1.3.1. Концепція секціонування

Розглянемо різні типи схем секціонування, які *Oracle* використовує для таблиць і індексів, і відсікання розділів.

Oracle застосовує чотири схеми секціонування таблиць:

- *Секціонування, засноване на діапазонах*. Дані поділяються на розділи за діапазоном значень. Це зазвичай використовується для дат. Наприклад, всі дані по «кварталу 1 цього року» поміщаються в один розділ, всі дані по «кварталу 2 цього року» - в інший і т.д.
- *Секціонування, засноване на випадковому виборі*. Хеш функція застосовується до колонки(ок) таблиці. Чим цей стовпець більш унікальний, тим краще. Первинний ключ, наприклад, є чудовим хеш-ключем. Хеш-функція

повертає число між 1 і *N* в залежності від того в якому із *N* розділів знаходяться дані. Хеш функція працює краще, коли *N* є ступенем 2, це обумовлюється алгоритмом випадкового розбиття, який *Oracle* використовує всередині.

- *Секціонування, засноване на списку*. Встановлюються дискретні списки значень для того, в якому розділі знаходяться дані. Наприклад, можна використовувати поле *CODE* і вказати, що записи з кодами *A, X* та *Y* переходять в розділ *P1*, а записи з кодами *B, C* і *Z* - в розділ *P2*. На відміну від двох попередніх схем, які можуть працювати з множиною столбців, секціонування, засноване на списку, обмежується єдиним стовпцем. Це означає, що буде проглядатися тільки один стовпець в таблиці при проведенні розбиття і розташування даних.
- *Складене секціонування.* Ця схема секціонування є гібридною. Дані поділяються за діапазонами. Потім кожен діапазон може бути розбитий або за допомогою випадкової схеми, або за допомогою списку. Наприклад, можна встановити 4 розділи за діапазонами, по одному на квартал року, і потім кожен поквартальний розділ розбити на вісім розділів, використовуючи хеш-функцію. В результаті вийде 4 x 8 = 32 розділи. Або ж можна встановити 4 поквартальних розділів, але потім розбити кожен поквартальний розділ на 4 регіони за допомогою списку, що дасть в результаті 4 x 4 = 16 розділів. Цікаво відзначити, що при складеному секціонуванні число розбитих розділів може бути неоднаковим. Таким чином, один з розділів може складатися з 4 підрозділів, а інший - з 5.

Для секціонування індексів в *Oracle* застосовуються два основні способи (див. рис.1.5.) :

- *Локально розділений індекс*. Для кожного із створених табличних розділів формується індексний розділ. Дані в кожному індексному розділі в точності вказують на дані в табличному розділі. Це говорить про те, що таблиця і індекс є різвнорозділеними. Існує взаємно однозначна відповідність поміж розділами таблиці і індекса, так що якщо в таблиці є *N* розділів, то і у індекса також буде *N* розділів.

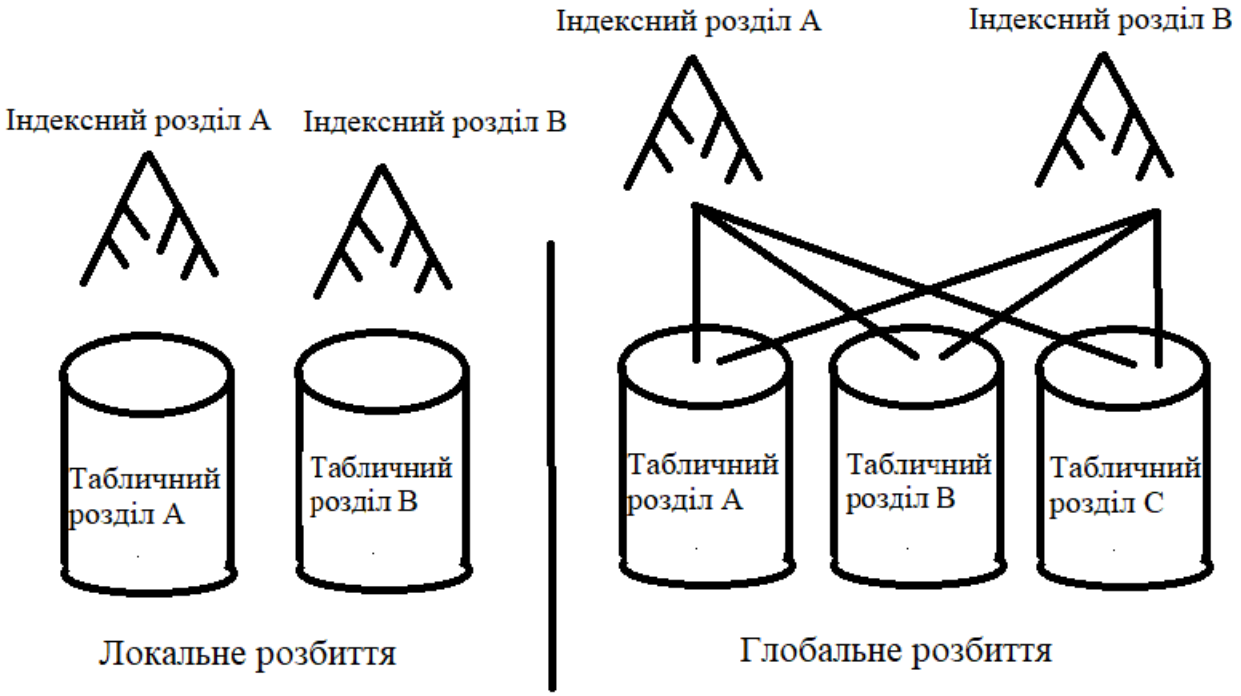

Рис. 1.5. Локально і глобально розділені індекси

- *Глобально розділений індекс*. Індекс розбивається по своїй схемі. Немає ніякої відповідності між розділами таблиці і індексу. Єдиний індексний розділ може вказувати на дані в будь-яких розділах таблиці. Глобально розділений індекс може бути побудований тільки за допомогою діапазонів. Індекси для розділених таблиць є за замовчуванням глобальними індексами в єдиному розділі (за замовчуванням вони взагалі не є розділеними індексами).

Таким чином, можна розбити таблицю за допомогою діапазонів, списка, хеш-функції або деякої комбінації цих методів. Індекси можуть бути розділені локально або глобально. Наступне поняття, яке буде розглянуто - це відсікання розділів.

1.3.2. Відсікання розділів

Виключення або відсікання розділів - це здатність оптимізатора виключати з розгляду конкретні розділи при оцінці запиту. Уявимо, що існує таблиця, розбита по стовпцю *SALE_DATE*. Дані за кожен місяць відносяться до одного розділу. І є локально розбитий по стовпцю *SALE_DATE* індекс. Надходить такий запит:

- *select * from t where sale_date = :x*

Цей запит може використовувати один із двух планів запиту в залежності від значень в стовпці *SALE_DATE*. Припустимо, що оптімізатор вибрав повне сканування таблиці. Так як таблиця розбита по стовпцю *SALE_DATE*, оптимізатор знає, що (*N-1*) розділів цієї табліці не містять необхідних даних. За визначенням, тільки в одному з розділів можуть міститися необхідні дані.

Таким чином, замість повного сканування вмісту таблиці *Oracle* проведе повне сканування єдиного розділу, суттєво скоротивши кількість вводів / виводів, які були б виконані в іншому випадку.

В якості альтернативного варіанту, припустимо, що план запиту включає сканування індексного діапазону. Оптимізатор не розглядатиме (*N-1*) індексних розділів. Йому відомо, що всі рядки, які нас цікавлять знаходяться в одному індексному розділі, так що необхідно сканувати єдиний індексний розділ, отримати *ROWID* рядків і потім забезпечити доступ до цього єдиного розділу для отримування даних.

Оптимізатор досить розумний, щоб не розглядати всі розділи. З точки зору продуктивності, це є однією з основних причин використання секціонування.

1.3.3. Причини використання секціонування

Секціонування використовується за трьома основними причинами:

1. Підвищення доступності
2. Спрощення адміністрування
3. Збільшення продуктивності

Я розташував ці пункти в порядку їх досяжності і значущості. Почнемо з найлегшою мети.

1. Підвищення доступності.

Існують два основні методи підвищення доступності системи задопомогою секціонування: виключення розділу і алгоритм розбиття.

Допустимо, що ми працюємо зі сховищем даних, і файли даних погано переміщуються (зіпсований носій). Необхідно повернутися до резервної копії та відновити її. Але один з кінцевих користувачів все ще працює з базою даних.

Оскільки було виконано секціонування і розділи були розміщені в окремих табличних простірах, фактично потрібно відновити тільки один розділ в одному табличному просторі. Це можна зробити в автономному режимі. Однак більшість даних не буде брати участь в операції відновлення, і кінцеві користувачі будуть працювати з тими ж самими таблицями і з тими ж самими запитами. Їх запити знають, що їм не потрібно звертатися до цих даних, їх видаляють з розгляду, і запити продовжують роботу. Відмова не відбувається. З цієї точки зору система стає більш доступною.

Система стає доступнішою при секціонуванні ще й тому, що доводиться мати справу з невеликими частинами системи. Допустимо, що система містить п'ятдесят розділів по 2 Гбайт і необхідно відновити один з них. Таким чином, виникає проблема не в 100 Гбайт. Кожна адміністративна задача, яка працює з об'єктом автономно (робить його недоступним), впливає лише на невеликий об'єкт, так що операції проходять швидше. Уявмо, що необхідно перебудувати бітовий індекс після пари місяців покрокового завантаження (бітові індекси погано відповідають на покрокові відновлення, їх необхідно перебудовувати на періодичній основі). Чому буде віддано перевагу: перебудові повного бітового індексу в одному великому операторі, що вимагає значних ресурсів, або перебудові однієї п'ятдесятої частини бітового індексу за один раз?

2. Спрощення адміністрування.

З невеликими об'єктами працювати легше. Наприклад, якщо раптом виясниться, що 50% рядків в таблиці є мігруючими, і необхідно це виправити, то виконати цю операцію буде набагато легше, якщо таблиця розділена. Для того щоб закріпити мігруючі рядки, необхідно перебудувати об'єкт, в даному випадку таблицю. Якщо є одна таблиця розміром 100 Гбайт, доведеться обробляти один дуже великий шматок, послідовно використовуючи *ALTER TABLE MOVE*. Якщо ж є 50 розділів по 2 Гбайт, то можна перебудувати розділи по одному. Крім того, якщо це робиться не в годину пік, можна виконувати оператори *ALTER TABLE MOVE PARTITION* паралельно в окремих сесіях, що потенціально скорочує час роботи. При використанні локально розділених індексів в цій розділеній таблиці операція по перебудові індексів займе також значно менше часу. Фактично все,

що можна робити з цільним об'єктом, можна застосувати до єдиного розділу в розділеному об'єкті.

Інший фактор, який відіграє певну роль у секціонуванні і адмініструванні - це концепція, яка називається «движущееся окно». Вона застосовується в сховищах даних і в деяких системах транзакцій, особливо для журналу аудиту, за допомогою якого система на протязі *N* років може в оперативному режимі зберігати інформацію. Допустимо, що у нас є інформація, заснована на часі - сховище даних відомостей про продажі за останній місяць або журнал аудиту, в якому фіксується час створення кожного запису. В кожному місяці хотілося б отримувати саму пізню дату, видаляти її і замінювати найновішою датою місяця. Якщо використовувати звичайну таблицю це перетвориться в жахливе видалення (*DELETE*) старих даних з подальшою вставкою (*INSERT*) нових даних. Процедури періодичного очищення за допомогою *DELETE* і завантаження за допомогою *INSERT* проковтнуть величезну кількість ресурсів (обидві генерують великий обсяг інформації щодо скасування та повторного виконання). Вони будуть намагатися самостійно розбивати індекси і таблиці. Операції видалення залишають дірки, які, може будуть, а може, і не будуть знову використовуватися через якийсь час, збільшуючи фрагментацію і руйнуючи простір.

З секціонуванням процес збирання старих даних спрощується. Він зводиться до виконання команди *ALTER TABLE*, яка видаляє розділ з найстарішими даними. Щоб ввести нові дані, потрібно завантажити і індексувати окрему таблицю, а потім додати її до існуючої розділеної таблиці за допомогою команди *ALTER TABLE ... EXCHANGE PARTITION*, яка виконує обмін між таблицею і розділом. В даному випадку простим додаванням даних до таблиці ще проводиться обмін поміж повною таблицею і пустим розділом. Якщо потурбуватися про те, щоб виключити підтвердження можлвих обмежень, то кінцевим користувачам не доведеться простоювати.

3. Збільшення продуктивності.

Як було показано вище, використання секціонування для збільшення продуктивності - це палиця двох кінців. Отримання додаткової, вимірюваної продуктивності за допомогою секціонування є найтяжчим завданням.

Секціонування повинно застосовуватися для вирішення проблем, з якими стикається користувач. Просте застосування секціонування, без будь-якого планування, з великою ймовірністю викличе погіршення показників продуктивності. Можна порівняти секціонування з медициною. Медицина здатна зробити хворих людей здоровішими, сильнішими і швидшими. Медицина може вбити здорову людину при неправильному лікуванні. Секціонування робить те ж саме з базою даних. Перед його застосуванням необхідно вивчити мету його використання. Чи вам потрібно позбутися від адміністративних накладних витрат? Або підвищити доступність? Або ж основною метою є збільшення продуктивності?

Висновки до розділу

Підводячи підсумки до тих пір поки система не перевантажена і немає необхідності в застосуванні певних засобів спільно використовуваного сервера, найкращим рішенням є робота в конфігурації виділеного сервера. Виділений сервер простіший в установці і налаштуванні. Є певні операції, які повинні бути виконані в режимі виділеного сервера, так що у кожної бази даних будуть встановлені і виділений і спільно використовуваний сервер, або тільки виділений сервер.

З іншої сторони, якщо відомо, що буде дуже велика кількість користувачів і тому потрібно впроваджувати спільно використовуваний сервер, наполегливо рекомендую виконати розробку і тестування з'єднань спільно використовуваного сервера. Імовірність відмови буде вище, якщо в розробці буде враховуватися тільки конфігурація виділеного сервера і не буде проведено тестування з застосуванням конфігурації спільно використовуваного сервера. Потрібно піддати систему навантаженню, провести оцінку продуктивності і упевнитися в тому, що система не монополізує занадто довго спільно використовувані сервери. Якщо всі ці проблеми будуть виявлені в процесі розробки, то виправити їх буде набагато легше, ніж на стадії введення. Можна використовувати деякі засоби, наприклад створити додаткові черги *AQ* для скорочення тривалих

процессов, але необхідно спроектувати це в додатку. Найкраще це робити в процесі розробки програми.

Якщо ж в додатку вже застосовується пул з'єднань відповідного розміру (наприклад, пул з'єднань *J2EE*), то в більшості випадків вибір спільно використовуваного сервера може негативно позначитися на продуктивності. Вже є розроблений пул з'єднань, який здатний в будь-який момент часу забезпечити одночасну роботу, і потрібно, щоб кожне із з'єднань було прямим з'єднанням виділеного сервера. Інакше вийде, що ви підключаєте один пул з'єднань до іншого пулу з'єднань.

Щодо кластеризації, якщо система повинна володіти високою доступністю або можливістю масштабування, то необхідно розглядати питання про використання кластеризації.

Секціонування може бути також приголомшливою штукою. Розбивка величезних проблем на безліч маленьких проблем (алгоритм розбиття) в деяких випадках може значно скоротити час обробки. Але секціонувавания може також бути жахливим, при невідповідному застосуванні воно здатне катастрофічно збільшити час обробки.

Потрібно точно знати, для чого застосовується секціонування. Не потрібно використовувати його до тих пір, поки у вас немає чіткої мети!

У багатьох випадках цілі, які необхідно досягти за допомогою секціонування, можна представити у вигляді такої схеми (рис. 1.6):

Рис. 1.6. Схема можливостей секціонування

Можна вибрати будь-яку точку на сторонах або всередині трикутника, але це означає, що, вибираючи одну точку, ми віддаляємося від двох інших. Перш за все потрібно з'ясувати, чому виникла необхідність у використанні секціонування, і потім розробити фізичні схеми. Наскільки великий цей трикутник і, отже, наскільки далеко один від одного ці точки, це повністю питання реалізації та потреб. В деяких випадках трикутник дуже маленький, тобто досяжні всі три мети. В інших випадках, можливо, доведеться пожертвувати однією метою на користь іншої.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ СТРУКТУР ПАМ'ЯТІ, ПАРАЛЕЛІЗМУ ТА УЗГОДЖЕНОГО ЧИТАННЯ

2.1. Структури пам'яті

В цьому розділі розглядаються три основних структури пам'яті *Oracle*.

- *Глобальна область системи* (*System Global Area-SGA)*. Це великий спільно використовуваний сегмент пам'яті, до якого буквально всі процеси *Oracle* будуть звертатися в той чи інший момент часу.
- *Глобальна область процесу* (*Process Global Area-PGA*). Це приватна область окремого процесу або потоку, недоступна з інших процесів або потоків.
- *Глобальна область користувача* (*User Global Area - UGA*). Ця область пам'яті пов'язана з конкретним сеансом. Вона буде розташована або в області *SGA* (якщо підключення до бази даних виконано за допомогою спільно використовуваного сервера), або в *PGA* (якщо підключення до бази даних виконано за допомогою виділеного сервера).

Спочатку ми розглянемо області *PGA* і *UGA*, а потім перейдемо до вивчення дійсно важливої структури - області *SGA*.

2.1.1. Глобальна область процесу і глобальна область користувача

Область *PGA* - це область, яка пов'язана з конкретним процесом . Іншими словами, це область пам'яті, яка пов'язана з окремим процесом або потоком операційної системи. Ця область пам'яті недоступна ніякому іншому процесові/потокові в системі. Як правило, вона виділяється за допомогою викликів *C*-функцій *malloc()* або *memmap()* часу виконання і може збільшуватися (або зменшуватися) під час виконання.

Область *PGA* ніколи не виділяється в області *SGA Oracle* - вона завжди виділяється процесом або потоком тільки локально.

Область *UGA*, по суті, представляє стан процесу. Сеанс завжди повинен мати доступ до цієї області пам'яті. Розташування області *UGA* повністю залежить від методу підключення до бази даних *Oracle*. Якщо підключення було виконано через спільно використовуваний сервер, *UGA* повинна зберігатися в структурі пам'яті, в якій кожен процес спільно використовуваного сервера має доступ – такою областю буде *SGA*. Таким чином, сеанс може застосовувати будь-який з спільно використовуваних серверів, оскільки будь-який з них може зчитувати і записувати дані сеансу. З іншого боку, при використанні підключення за допомогою спільно використовуваного сервера потреба в універсальному доступі до стану сеансу відпадає, і *UGA* стає повністю аналогічною області *PGA*. Фактично вона і буде міститися в області *PGA* спільно використовуваного сервера. Якщо поглянути на статистичну інформацію системи, легко переконатися, що в режимі виділеного сервера область *UGA* зберігається в області *PGA* виділеного сервера (розмір *PGA* буде більшим або дорівнюватиме розміру області *UGA*: розмір *PGA* буде враховувати також розмір *UGA*).

Таким чином, область *PGA* містить пам'ять, виділену для процесу, і може містити область *UGA*. Решта областей пам'яті *PGA* в основному служать для виконання операцій сортування, злиття бітових індексів і хешування. Можна сміливо стверджувати, що поряд з *UGA* ці процеси вносять найбільший внесок у формування області *PGA*.

Починаючи з *Oracle9i Release 1* і наступних версій, існують два способи управління цими областями пам'яті, відмінними від *UGA*, в області *PGA*.

- *Управління пам'яттю PGA вручну*, при якому СУБД *Oracle* вказує, який обсяг пам'яті вона може використовувати для виконання операцій сортування і хешування в ході конкретного процесу.
- *Автоматичне управління пам'яттю PGA*, при якому СУБД *Oracle* вказують, який обсяг пам'яті вона повинна намагатися використовувати на рівні всієї системи.

Способи виділення та використання пам'яті в кожному з цих випадків дуже різняться, тому ми розглянемо їх по черзі. Мушу зазначити, що в середовищі *Oracle9i* при використанні підключення за допомогою спільно використовуваного сервера можна застосовувати тільки управління пам'яттю *PGA* вручну. Це обмеження було знято в *Oracle10g Release 1* (і наступних версіях). У цій версії бази даних при використанні підключень за допомогою спільно використовуваного сервера можна застосовувати, як автоматичне керування пам'яттю *PGA*, так і управління вручну.

Метод управління пам'яттю *PGA* визначається параметром ініціалізації бази даних *WORKAREA_SIZE_POLICY*, який можна змінювати на рівні сеансу. У версіях *Oracle9i Release 2* і наступних за замовчуванням цим параметром ініціалізації присвоюється значення *AUTO*, яке вказує про необхідність використання автоматичного управління пам'яттю *PGA*, коли це можливо. У *Oracle9i Release 1* значення цього параметра за замовчуванням - *MANUAL* (вручну).

2.1.2. Управління пам'яттю *PGA* вручну

При правлінні пам’яттю *PGA* вручну найбільший вплив на розмір області *PGA* без урахування об’єму пам'яті, виділеного сеансом для таблиць *PL / SQL* і інших змінних, надаватимуть наступні параметри.

- *SORT_AREA _SIZE*. Цей параметр визначає загальний обсяг пам'яті, який буде використаний для сортування інформації перед її скиданням на диск.
- *SORT_AREA_RETAINED_SIZE*. Цей параметр визначає обсяг пам'яті, який буде використаний для зберігання відсортованих даних по завершенні операції сортування. Тобто, якщо значення параметра *SORT_AREA _SIZE* - 512 Кбайт, а параметра *SORT_AREA_RETAINED_SIZE* - 256 Кбайт, процес сервера буде використовувати до 512 Кбайт пам'яті для сортування даних під час початкової обробки запиту. По завершенні сортування область сортування буде "стиснута" до 256 Кбайт, а будь-які дані сортування, які не вмістилися в цьому обсязі, будуть записані в тимчасовий табличний простір.

- *HASH_AREA_SIZE*. Цей параметр визначає обсяг пам'яті, який процес сервера повинен використовувати для зберігання хеш-таблиць. Ці структури використаються під час хеш-з'єднань - як правило, при об'єднанні більшого набору з іншим набором. Менший з двох наборів буде хешуватися в пам'ять, а всі дані, що не помістилися в хеш-області пам'яті, будуть збережені в тимчасових табличних просторах, визначених ключем об'єднання.

Описані параметри керують об'ємом пам'яті, який СУБД *Oracle* буде використовувати для виконання сортування або хешування даних перед їх записом (скиданням) на диск, і обсяг цього сегмента пам'яті, який буде збережений після виконання сортування. В основному, обсяг пам'яті *SORT_AREA_SIZE SORT_AREA_RETAINED_SIZE* виділяється з області *PGA*, а обсяг пам'яті, заданий параметром *SORT_AREA_RETAINED_SIZE*, буде розміщуватися в області *UGA*. Інформацію про поточний стан використання пам'яті *PGA* і *UGA* можна отримати і відслідковувати, запитуючи спеціальні V$-уявлення *Oracle*, звані також динамічними уявленнями продуктивності.

2.1.3. Автоматичне управління пам'яттю *PGA*

У версії, починаючи з *Oracle9i Release 1*, був включений новий спосіб управління і використання пам'яттю PGA, що виключає використання параметрів *SORT_AREA_SIZE, BITMAP_MERGE_AREA_SIZE* і *HASH_AREA_SIZE*. Він був введений для вирішення перерахованих нижче проблем.

- *Спрощення використання.* Необхідність правильної установки параметрів **_AREA_SIZE* породжувала безліч питань. Крім того, існувала велика плутанина в уявленнях про реальну роботу цих параметрів і про реальне виділення пам’яті.

- *Управління пам'яттю вручну було методом "одного розміру на всі випадки життя".* Як правило, по мірі збільшення кількості користувачів, одночасно виконуючих одні і ті ж додатки стосовно баз даних, обсяг пам'яті, використовуваної для сортування/хешування, зростав в лінійній пропорції. Якщо 10 одночасно працюючих користувачів областю сортування розміром 1 Мбайт

використовували 10 Мбайт пам'яті, то 100 одночасно працюючих користувачів, ймовірно, будуть використовувати 100 Мбайт, 1000 користувачів - 1000 Мбайт і так далі. Якщо тільки адміністратор бази даних не сидить за консоллю, постійно змінюючи розміри області сортування/хешування, швидше за все, протягом усього робочого дня всі користувачі будуть застосовувати одні і ті ж значення. Згадайте, як у попередньому прикладі кількість операцій фізичного введення-виведення в часовому просторі зменшувалася зі збільшенням дозволеного для використання обсягу ОЗУ. Якщо ви самостійно виконайте це завдання, то скоріш за все помітите зменшення часу відгуку зі збільшенням обсягу ОЗУ, доступного для сортування. Виділення пам'яті вручну веде до фіксування обсягу пам'яті, який буде використаний для сортування, рівним більш-менш постійному значенню, незалежно від дійсно доступного обсягу пам'яті. Автоматичне управління пам'яттю дозволяє користуватися тією пам'яттю, яка доступна - цей метод управління динамічно налаштовує обсяг використовуваної пам'яті в відповідності з робочим навантаженням.

- *Управління пам'яттю.* Вище описана обставина робила важким, якщо не неможливим, утримання екземпляра *Oracle* в "рамках розумного розподілу пам'яті". Об'ємом пам'яті, який екземпляр збирався використати з, неможливо було керувати, оскільки не існувало ніякого реального засобу управління кількістю одночасних операцій сортування/хешування, які виконуються. Занадто легко можна було виділити для використання більше реальної пам'яті (реальної фізичної пам'яті),ніж було доступно на комп'ютері.

Допустимо до розгляду автоматичного управління пам'яттю *PGA*. Спочатку потрібно просто встановити розмір області *SGA*. Ця область - сегмент пам'яті фіксованого розміру, тому її розмір можна визначити дуже точно. Причому отримане значення буде представляти розмір всієї цієї області (до тих пір, поки він не буде змінений). Потім ви вказуєте базі даних *Oracle* наступне: цим обсягом пам'яті потрібно спробувати обмежити всі робочі області - він буде служити свого роду новою "парасолькою" для використовуваних областей сортування і хешування. Після цього на комп'ютері з 2 Гбайт фізичної пам'яті

теоретично можна виділити 768 Мбайт пам'яті області *SGA* і 768 Мбайт - області *PGA*, залишаючи 512 Мбайт для ОС і інших процесів. Я сказав "теоретично", оскільки насправді не все так просто, хоча реальна ситуація і буде наближатися до описаної. Перш ніж приступити до розгляду того, чому це відбувається, розглянемо процес налаштування автоматичного управління пам’яттю *PGA* і його включення.

Процес настройки цього управління пам'яттю включає в себе вибір правильних значень двох параметрів ініціалізації екземпляра, а саме:

- *WORKAREA_SIZE_POLICY*. Значення цього параметра може бути встановлено рівним або *MANUAL*, при якому для управління обсягом виділеної пам'яті будуть використані параметри розмірів областей сортування і хешування, або *AUTO*, при якому обсяг виділеної пам'яті буде змінюватися в залежності від поточного робочого навантаження бази даних. Використане за замовчуванням і рекомендоване значення цього параметра - *AUTO*.
- *PGA_AGGREGATE_TARGET*. Цей параметр керує сумарним об'ємом пам’яті, який екземпляр повинен виділяти для всіх робочих областей, використаних для виконання сортування/хешування даних. Його значення, яке встановлюється за замовчуванням, залежить версії і може змінюватися за допомогою різних утиліт, таких як *DBCA*. У більшості випадків при використанні автоматичного управління пам’яттю *PGA* значення цього параметра необхідно встановлювати явно.

Отже, якщо вважати, що в якості значення параметра *WORKAREA_SIZE_POLICY* вибрано *AUTO*, а параметр *PGA AGGREGATE_TARGET* має нульове значення, це буде означати застосування нового методу автоматичного управління пам'яттю PGA. Цей режим можна "включити" за допомогою команди *ALTER SESSION* в сеансі або на системному рівні.

Таким чином, основна мета застосування автоматичного управління пам’яттю *PGA* - максимізація використовуваного обсягу ОЗУ при одночасному запобіганню застосування більшого обсягу ОЗУ, ніж потрібно. При управлінні

пам’яттю вручну ця мета була буквально недосяжна. При встановленні значення *SORT_AREA_SIZE* рівним 10 Мбайт один користувач, що виконує операцію сортування, задіяв би до 10 Мбайт пам'яті під область сортування. Якби 100 користувачів виконували таку ж операцію, вони зайняли б до 1000 Мбайт пам'яті. При наявності в системі 500 Мбайт вільної пам'яті єдиний користувач, виконуючий сортування, міг би зайняти значно більший обсяг пам’яті, а 100 користувачів змушені були б задовольнятися значно меншим обсягом. Саме для вирішення таких проблем і було розроблено автоматичне управління пам'яттю *PGA*. При незначному робочому навантаженні використання пам'яті може бути максимальним, а в міру збільшення навантаження і збільшення кількості користувачів, що виконують операції сортування або хешування, обсяг виділеної для них пам'яті буде зменшуватися, забезпечуючи використання всього доступного обсягу ОЗУ при одночасному запобіганню спроб запиту великого обсягу пам'яті, ніж існує фізично.

2.1.4. Вибір між ручним і автоматичним управлінням пам'яті

Так який же метод слід застосовувати: ручний чи автоматичний? Особисто я віддаю перевагу за замовчуванням використовувати автоматичне керування пам'яттю *PGA*.

Однією з найбільш неприємних завдань адміністратора бази даних може бути установка індивідуальних параметрів, особливо таких, як *SORT | HASH_AREA_SIZE* і так далі. Мені багато разів доводилося стикатися з системами, працюючими з неймовірно низькими значеннями цих параметрів - настільки низькими, що це призводило до дуже великого зниження продуктивності системи. Ймовірно, це пояснюється тим, що значення цих параметрів, встановлені за замовчуванням, дуже малі: 64 Кбайт для сортування та 128 Кбайт для хешування. Існує безліч суперечливих думок з приводу оптимальних значень цих параметрів. Крім того, оптимальні значення можуть змінюватися протягом робочого дня. О 8 годині ранку за наявності двох користувачів сфері сортування, рівний 50 Мбайт, виділений для одного зареєстрованого користувача, може бути цілком допустимим. Але опівдні, при

наявності 500 користувачів, значення 50 Мбайт може виявитися зовсім недопустимим. Саме в цій ситуації доцільно використовувати параметри *WORKAREA_SIZE_POLICY=AUTO* і *PGA_AGGREGATE_TARGET*. З методичної точки зору установка значення *PGA_AGGREGATE_TARGET* - обсягу пам'яті, який СУБД *Oracle* повинна мати право використовувати для виконання операцій сортування і хешування - простіше, ніж спроби підбору ідеальних значень параметрів *SORT | HASH_AREA_SIZE*, тим більше, що таких ідеальних значень просто не існує. Оптимальні значення залежать від конкретного робочого навантаження.

За історично сформованою традицією адміністратори баз даних конфігурували обсяг пам'яті, який використовується Oracle, встановлюючи розмір області *SGA* (що складається з кешу буферів, журнального буфера і пулів *Shared, Large* і *Java*). Пам'ять, що залишилася вільною, повинна була використовуватися виділеним і спільно використовуваним серверами в області *PGA*. Адміністратор бази даних володів дуже незначними можливостями управління використанням обсягу цієї пам’яті. Він міг визначити параметр *SORT_AREA_SIZE*, але в цьому випадку при одночасному виконанні 10 операцій сортування СУБД *Oracle* могла б використовувати до 10 * *SORT_AREA_SIZE* байт ОЗУ. При паралельному виконанні 100 операцій сортування Oracle використовувала б 100 * *SORT_AREA_SIZE* байт, при 1000 операціях - 1000 * *SORT_AREA_SIZE* і так далі. З огляду на те, що в *PGA* поміщаються і інші об'єкти, адміністратор позбавлений реальної можливості управління максимальним використанням пам'яті *PGA* системи.

Бажано, щоб використання цієї пам'яті регулювалося реальною потребою. Чим більше користувачів, тим менший обсяг ОЗУ кожен з них повинен використовувати, а чим їх менше, тим більший обсяг пам'яті повинен бути доступним для застосування кожним з них. Установка параметра *WORKAREA_SIZE_POLICY=AUTO* якраз і є способом досягнення цієї мети. Тепер адміністратор бази даних може вказати єдиний розмір *PGA_ AGGREGATE_TARGET*, який представляє собою максимальний обсяг пам'яті *PGA*, який база даних повинна прагнути використовувати. *Oracle* буде

розподіляти цю пам'ять між нормативними сеансами так, як вважає за потрібне. Більше того, *Oracle9i Release 2* і наступні версії надають консультативну довідку з налаштування *PGA* (яка являється частиною пакету *Statspack*, доступною за динамічне представлення інформації продуктивності *V$* і відображену у вікні *Enterprise Manager*), подібну консультативній довідці з налаштування кеша буферів. Згодом довідка підкаже, яке значення *PGA_AGGREGFATE_TARGET* оптимально для мінімізації кількості фізичних операцій введення-виведення в тимчасових таблиних просторах даної системи. Цю інформацію можна використовувати або для динамічного інтерактивного зміни розміру *PGA* (при наявності достатнього очного обсягу ОЗУ), або для прийняття рішення про те, чи потребує сервер в додатковому обсязі ОЗУ для досягнення оптимальної продуктивності.

Однак бувають ситуації, в яких небажано використовувати автоматичне управління пам'яттю. Цей метод був розроблений для багатокористувацького середовища. В ситуаціях, коли передбачається приєднання нових користувачів до системи, автоматичне керування пам'яттю буде обмежувати обсяг виділеної пам'яті лише певною частиною обсягу, заданого параметром *PGA_AGGREGATE_TARGET*. Але як бути, якщо потрібно задіяти весь доступний обсяг пам'яті? Що ж, в цьому випадку саме час скористатися командою *ALTER SESSION* для відключення автоматичного управління пам'яттю в конкретному сеансі (залишаючи його активізованим для всіх інших) і установки вручну потрібних значень параметрів *SORT_HASH_AREA_SIZE*. Наприклад, як вчинити в разі масштабного процесу пакетної обробки, що виконується о 2 годині ночі і виконує великі хеш-об'єднання, побудови ряду індексів тощо? Цьому процесові не потрібно "соромитися" в плані використання пам'яті - йому потрібна вся доступна пам'ять, оскільки в даний момент він являється єдиним, який виконуються в базі даних. Цілком очевидно. що це пакетне завдання може видавати команду *ALTER SESSION* і використовувати всі доступні ресурси системи.

Іншими словами, я вважаю за краще застосовувати автоматичне керування пам’яттю для сеансів кінцевих користувачів - для додатків, які регулярно

виконуються в базі даних. Застосовувати управління пам'яттю вручну доцільно при виконанні великих пакетних завдань, що функціонують в ті періоди часу, коли вони є єдиними активними процесами.

2.1.5. Висновок з питань використання областей *PGA* і *UGA*

Отже, ми розглянули дві структури пам'яті: *PGA* і *UGA*. Тепер ми усвідомили, що область *PGA* є приватною для процесу. Вона представляє собою набір змінних, які виділені або спільно використовуваний сервер Oracle повинен зберігати незалежно від сеансу. Область *PGA* - це "хмара" пам’яті, в якій можуть розподілятися інші структури. *UGA* також є "хмарою" пам’яті, в якій можуть бути визначені різні структури, характерні для сеансу. Область *UGA* виділяється з області *PGA* при підключенні до бази даних через виділений сервер і з області *SGA* при підключенні через спільно використовуваний сервер. З цього випливає, що при використанні спільно використовуваний сервер необхідно визначити розмір пулу *Large* області *SGA* так, щоб забезпечити достатній обсяг для підключення до бази даних будь-якої можливої кількості одночасно працюючих користувачів. Тому в загальному випадку область *SGA* бази даних, яка підтримує підключення за допомогою спільно використовуваного сервера, значно більше області *SGA* аналогічно сконфігурованій базі функціонуючій тільки в режимі виділеного сервера.

2.2. Системна глобальна область

Кожен екземпляр *Oracle* володіє однією великою структурою пам'яті, так званою системною глобальною областю (*System Global Area - SGA*). Це велика структура пам'яті спільного використання, до якої в той чи інший момент часу буде звертатися кожен процес *Oracle*. Її розмір буде змінюватися від декількох мегабайт в невеликих тестових системах до сотень мегабайт в середніх і великих системах і до багатьох гігабайт в дійсно великих системах.

В операційній системі *UNIX* область *SGA* представляє собою фізичний компонент, який можна "бачити" з командного рядка ОС. Фізично вона

реалізована у вигляді сегмента пам'яті спільного використання - сегмента пам'яті, до якого можуть підключатися процеси. Можлива ситуація, коли *SGA* буде існувати в системі без єдиного процесу *Oracle* – пам’ять виявиться надана самій собі. Однак слід зазначити, що наявність області *SGA* без будь-яких процесів Oracle свідчить про збої бази даних з будь-якої причини. Ця ситуація незвичайна, але вона може зустрічатися.

У середовищі *Windows* неможливо побачити *SGA* в якості окремого компоненту, як це можливо в системах *UNIX/Linux*. Оскільки на платформі *Windows Oracle* діє в якості єдиного процесу з єдиним адресним простором, область *SGA* виділяється процесу oracle.exe, як приватній області пам'яті. Диспетчер завдань *Windows* або інший засіб управління продуктивністю дозволяє з'ясувати обсяг пам'яті, виділений процесу oracle.exe, але при цьому не можна порівняти *SGA* з будь-якою іншою частиною виділеної пам'яті.

Всередині самого середовища Oracle інформацію про *SGA* можна переглядати незалежно від платформи, використовуючи ще одне "магічне" *V$*-представлення – *V$SGASTAT*.

Область *SGA* розбита на різні пули.

- *Пул Java.* Це фіксований обсяг пам'яті, виділений для віртуальної машини *Java (Virtual Java Machine - JVM)*, що діє в базі даних. У *Oracle10g* пул Java може оперативно змінюватися під час роботи бази даних.
- *Пул Large (Великий).* Цей пул використовується підключеннями до пам'яті сеансу за допомогою спільно використовуваного сервера, функціями, які паралельно виконуються для зберігання буферів повідомлень і засобом резервного копіювання *RMAN* для зберігання буферів дискового введення-виведення.
- *Пул Shared (Спільного використання).* Цей пул містить спільно використовувані курсори, збережені процедури, об'єкти стану, кеші словника даних і безліч інших об'єктів. Розміри цього пулу можуть оперативно змінюватися як в *Oracle10g*, так і в *Oracle9i*.
- *Пул Steams (Потоковий).* Цей пул пам'яті використовується виключно утилітою *Oracle Streams* (Потоки *Oracle*), призначеною для забезпечення

сумісного використання даних усередині бази даних. Цей пул був введений в версії *Oracle10g* і його розмір може оперативно змінюватися. Якщо пул *Streams* не сконфігурований, а функціональні можливості утиліти Steams активні, *Oracle* використовує до 10% обсягу цього пулу для резервування пам'яті для потоків.

- *Пул "Null" ("Нульовий").* Насправді цей пул не має імені. Він представляє собою пам'ять, призначену для зберігання буферів блоків (кешованих блоків бази даних), буфера журналу повторення і області «фіксованої *SGA*».

Типова область SGA може виглядати, як показано на рис. 2.1.

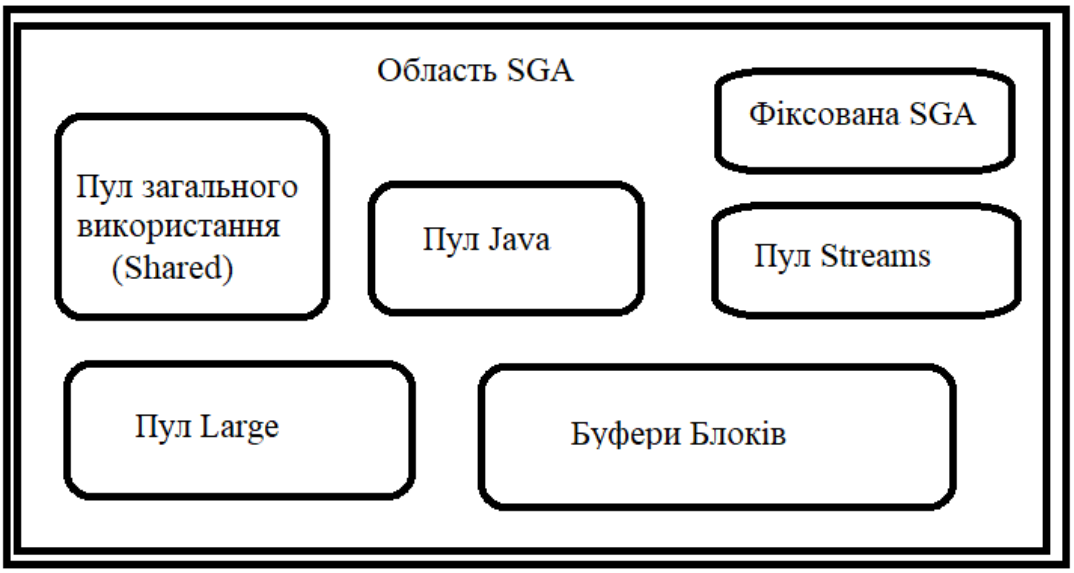

Рис.2.1. Типова область *SGA*

Найбільший вплив на загальний розмір області *SGA* надають такі параметри:

- *JAVA_POOL_SIZE*. Цей параметр керує розміром пулу Java.
- *SHARED_POOL_SIZE*. Цей параметр певною мірою керує розміром пулу Shared.
- *LARGE_POOL_SIZE*. Цей параметр керує розміром пулу *Large*.
- *DB_*_CACHE_SIZE*. Вісім з цих параметрів *CACHE_SIZE* керують розмірами різних доступних кешей буферів.

- *LOG_BUFFER*. Цей параметр певною мірою управляє розміром буфера повторення.
- *SGA_TARGET*. В *Oracle10g* і наступних версіях цей параметр використовується спільно з автоматичним управлінням пам'яттю SGA.
- *SGA_MAX_SIZE*. Цей параметр використовується для управління максимальним розміром, який може бути виділений для *SGA* під час роботи бази даних.

В *Oracle9i* різні компоненти *SGA* вимагають настройки їх розмірів адміністратором бази даних, але починаючи з *Oracle10g*, з'явилася нова можливість автоматичного керування пам'яттю *SGA*, при якому екземпляр бази виділятиме і перерозподілятиме пам'ять для різних компонентів у під час роботи відповідно до мінливого робочого навантаження. При використанні автоматичного управління пам'яттю в *Oracle10g* досить встановити значення параметра *SGA_TARGET* рівним бажаного розміру області *SGA*, надавши настройку всіх інших параметрів *SGA* самій базі даних. На основі цього значення екземпляр бази даних буде виділяти пам'ять для різних пулів в міру необхідності і навіть перерозподіляти пам'ять між пулами.

Незалежно від того, використовується автоматичне чи ручне управління пам’яттю, пам'ять різних пулів виділяється блоками, званими *гранулами.*

Одна гранула - це область пам'яті, розміром 4, 8 або 16 Мбайт. Гранула являється найменшим блоком виділення пам'яті, тому якщо надіслати запит для пулу *Java* 5 Мбайт при розмірі гранули 4 Мбайт, в дійсності *Oracle* виділить для пула *Java* 8 Мбайт пам'яті (8 - найменше ціле число, більше або рівне 5 і кратне розміру гранули рівному 4). Розміри гранули визначається розміром *SGA* (в деякій мірі це ствердження виглядає рекурсивним, оскільки розмір *SGA* залежить від розміру гранул). Розміри гранул використовуваних для кожного пула, можна з'ясувати, запитуючи уявлення *V$SGA_DYNAMIC_COMPONENTS*.

2.2.1. Фіксована *SGA*

Фіксована *SGA* - компонент *SGA*, розмір якого залежить від платформи і версії. Ця область компілюється в власне двійковий модуль *Oracle* під час установки (звідси і назва "фіксована"). Фіксована *SGA* содержит набір змінних, які вказують на інші компоненти *SGA*, і змінних, що містять значення різних параметрів. Розмір фіксованого *SGA* недоступний управлінню з боку адміністратора бази даних або програміста і, як правило, він дуже малий. Цю область можна вважати "загркзочным" розділом *SGA* - областю, яку *Oracle* використовує внутрішньо для відшукання інших елементів і фрагментів *SGA*.

2.2.2. Буфер повторення

Буфер повторення - область тимчасового, перед записом на диск, кешурования даних, які повинні бути записані в оперативні журнали повторення. Оскільки передача пам'ять-пам'ять виконується значно швидше передачі пам'ять-диск, застосування буфера журналу повторення може прискорити роботу баз даних. Дані будуть зберігатися в буфері повторення не дуже довго. Фактично, процес *LGWR* (процес запису в журнал) ініціює скидання на диск даних цієї області у перерахованих нижче випадках.

- Кожні три секунди.
- Коли хто-небудь виконує фіксацію транзакції.
- Коли процес *LGWR* отримує запит на перемикання журнальних файлів.
- Коли буфер повторення заповнюється на одну третину або обсяг містяться в ньому кешованих даних журналу повторення складає 1 Мбайт.

З цих причин лише в дуже рідкісних системах збільшення розміру буфера повторення більш ніж до декількох мегабайт надає якісь переваги. У той же час великі системи з безліччю одночасно виконуючих транзакцій можуть отримувати деякі переваги від застосування великих буферів журналів повторення, оскільки поки процес *LCWR* (відповідальний за скидання буфера журналу повторення на диск) записує частину журнального буфера, інші сеанси можуть його заповнювати. У загальному випадку використання великого, а не звичайного журнального буфера надає певні переваги довготривалої транзакції, яка генерує

безліч даних журналу повторення, оскільки вона буде постійно заповнювати частину буфера журналу повторення, поки процес *LCWR* зайнятий записом на диск іншій частині цього буфера. Чим більша і триваліша транзакція, тим більше переваг вона може отримати від більшого журнального буфера.

Заданий за замовчуванням розмір буфера повторення, керований параметром *LOG_BUFFER*, дорівнює більшому із значень 512 Кбайт і (128 * кількість процесорів) Кбайт. Мінімальний розмір цієї області залежить від ОС. Для з'ясування розміру цієї області досить встановити значення параметра *LOG_BUFFER* який дорывнює 1 байту і перезапустити базу даних.

2.2.3. Кеш буфера блоків

Досі було розглянуто порівняно невеликі компоненти області *SGA*. Теперь належить розглянути компонент, розмір якого може бути дуже великим. У кеші буфера блоків *Oracle* зберігає блоки бази даних перед їх записом на диск і після їх зчитування з диска. Ця область *SGA* особо важлива. Якщо вона буде занадто малою, запити будуть виконуватися нескінченно довго. Якщо ж вона буде занадто великою, це "посадить на голодний пайок" інші процеси (тобто у виділеного сервера не залишиться достатнього об'єма пам'яті для створення його області *PGA*, і ми навіть не зможемо запустити базу даних).

У попередніх версіях *Oracle* існував єдиний кеш буфера блоків, і все блоки з будь-яких сегментів поміщалися в цю єдину область. Починаючи з *Oracle* 8.0, для зберігання кешованих блоків з різних сегментів *SGA* можна використовувати три області.

- *Пул за замовчуванням* (*default pool*). Область звичайного кешування блоків з усіх сегментів. Ця область представляє первинний – і раніше єдиний – пул буфера.
- *Утримуючий пул* (*keep pool*). Альтернативний буферний пул, в якому за угодою будуть зберігатися часто використовувані сегменти, переміщені з буферного пулу за замовчуванням в зв'язку з необхідністю звільнення місця для інших сегментів.

• *Рецикльований пул* (*recycle pool*). Альтернативний буферний пул, в якому за угодою будуть зберігатися великі, дуже рідко використовувані сегменти, що призводять до тривалого скидання даних на диск, але не які не надають ніяких особливих переваг, оскільки з того моменту, коли блок буде потрібен знову, він вже буде видалений з кеша через його «застарівання». Ці сегменти можна виділяти із сегментів пулу за замовчуванням і утримуючого пулу для запобігання їх повного видалення з кеша через закінчення певного часу зберігання.

Зверніть увагу, що при описі утримуючого пулу і рециклюючого пулу було використано вираз "за згодою". Насправді ніщо не потребує використовувати ці пули тільки так, як описано. Фактично всі три пулу керують блоками майже ідентичним чином. Застосовувані ними алгоритми видалення внаслідок старіння або кешування блоків практично однакові. Мета використання цих пулів – це можливість поділу адміністратором сегментів на «гарячі, теплі і не заслуговуючі кешування області». Теоретично об'єкти в пулі за замовчуванням повинні були бути досить "гарячими" (тобто використовуватися досить часто), щоб гарантовано залишатися в кеші при будь-яких обставинах. Кеш буде зберігати їх в пам'яті внаслідок їх дуже високої популярності. Деякі сегменти можуть бути досить популярними, але не "гарячими" в повному сенсі. Ці блоки будуть вважатися "теплими". Вони можуть скидатися з кешу на диск для звільнення місця для деяких блоків, які застосовуються не настільки часто («не заслуговуючих кешування" блоків). Для збереження цих "теплих" блоків в кеші можна виконати одну з нижче описаних дій.

• Присвоїти ці сегменти утримуючому пулу для продовження їх перебування в буферному кеші.

• Присвоїти "не заслуговуючим кешування" сегменти рецикльованому пулу, зберігаючи при цьому розмір цього пулу досить маленьким, щоб операції приміщення блоків в кеш і їх видалення з нього виконувалися досить часто (це дозволить знизити накладні витрати, пов'язані з управлінням усіма цими блоками).

В результаті обсяг роботи з управління, яку повинен виконувати адміністратор бази даних, збільшиться, оскільки йому доведеться дбати про три

кеші, визначати їх розміри і розподіляти об'єкти між ними. Слід також пам'ятати, що не існує ніяких механізмів обміну пам'яттю між цими областями. Тому, якщо утримуючий пул містить великий обсяг невикористовуваного простору, він не зможе передати його перевантаженому пулу за замовчуванням і рециклюваному пулу. Але, як би там не було, в цілому ці пули представляють собою дуже ефективні засоби низкорівневої настройки, до яких, слід вдаватися після застосування всіх інших засобів (якщо запит можна переписати так, щоб він виконував тільки одну десяту частину операцій введення-виедення я вважатиму кращу цю можливість налаштування декількох буферних пулів!)

Починаючи з *Oracle9i*, в додаток до пулу за замовчуванням, утримуючому і рецикльованому пулу, адміністратори баз даних отримали в своє розпоряджження ще чотири не обов'язкових для використання кеша *db_Nk_caches*. Ці кеші були додані для підтримки в базі даних декількох розмірів блоків. В версіях, що передували *Oracle9i*, база даних повинна була використовувати тільки один розмір блоку (як правило, який дорывнював 2, 4, 8, 16 або 32 Кбайт). Починаючи з *Oracle9i*, база даних може застосовувати розмір блоку, заданий за замовчуванням, який задає розмір блоків, що зберігаються в пулі за замовчуванням, утримуючому і рецикльованому пулах, і до чотирьох розмірів блоків, що відрізняються від заданою ного за замовчуванням.

Управління блоками в цих буферних кешах виконується так само, як і управління блоками в первісному пулі за замовчуванням - використовувані при цьому алгоритми не зазнали ніяких змін.

2.2.3. Пул *Shared*

Пул *Shared* (пул спільного використання) - один з найбільш важливих сегментів області *SGA*, особливо в плані продуктивності і масштабованості. Занадто маленький пул *Shared* може привести до такого зниження продуктивності, що система буде здаватися "завислою". Занадто великий пул *Shared* може привести до аналогічного ефекту. Неправильне використання цього пулу може також бути небезпечним.

Так що ж собою являє пул *Shared*? В цій галузі *Oracle* кешує безліч "програмних" даних. При виконанні розбору запиту проаналізоване уявлення кешується в цій області. Перш ніж приступити до розбору всього запиту, СУБД *Oracle* виконує пошук в пулі *Shared* для з'ясування того, чи не була вже виконана ця задача. Виконуваний код *PL / SQL* також кешується в пулі *Shared*, щоб при наступному виконанні СУБД *OracIe* не довелося знову зчитувати його з диска. При наявності 1000 сеансів, виконуючих один і той же код, тільки одна копія коду завантажується і спільно використовується всіма сеансами. У цьому сегменті *Oracle* зберігає системні параметри. Тут же зберігається кеш словника (кешована інформація про об'єкти бази даних). Простіше кажучи, в пулі *Shared* зберігається буквально все.

Пул *Shared* характеризується безліччю невеликих (об'ємом по 4 Кбайт або менше) сегментів пам'яті. Слід мати на увазі, що 4 Кбайт не є жорсткою межею мета полягає в застосуванні невеликих сегментів пам'яті для запобігання фрагментації, яка відбувалася б, якби пам'ять виділялася сегментами, розміри яких сильно різняться – від дуже маленьких до дуже велких. Управління пам'яттю в пулі *Shared* здійснюється на основі застосування алгоритму *LRU* (*Least Recently Used algorithm* – алгоритм видалення сторінок, які найдовше не використовувалися). У цьому сенсі він аналогічний буферному кешу - якщо його не використовувати, його вміст втрачається. Щоб змінити цю поведіну можна використовувати службовий пакет *DBMS_SHARED_POOL*, який дозволяє примусово закріплювати об'єкти в пулі *Shared*. Цю процедуру можна застосовувати для загрузки часто використовуваних процедур і пакетів під час начального запуску бази даних, забезпечивши при цьому неможливість іх старіння. Однак у звичайних умовах, якщо якась час частина вмісту пам'яті пулу *Shared* не використовується повторно, це вміст застаріває. Навіть код *PL / SQL*, який може бути досить великим, управляється механізмом сторінкової організації так, що при виконанні коду дуже великого пакета лише дійсно необхідний код завантажується в пул *Shared* невеликими порціями. Якщо його не використовувати протягом тривалого періоду часу, цей код застаріє і буде

видалений з пулу *Shared* в разі його заповнення і необхідності звільнення місця для інших об'єктів.

Найпростіший спосіб руйнування пулу *Shared Oracle* - відмова від використання змінних зв'язування. Відмова від змінних зв’язування може "поставити систему на коліна" з двох причин.

- Система витрачає неприпустимо велику частину часу процесора для розбору запитів. .
- Система використовує занадто багато ресурсів для управління об'єктами в пулі *Shared* через відсутність повторного використання запитів.

Якби кожен представлений базі даних *Oracle* запит був унікальним запитом з жорстко закодованими значеннями, концепція пулу *Shared* понесла б суттєвої шкоди. Ця область пам'яті була розроблена з розрахунком на багаторазове використання планів запитів. Якщо кожен запит є суттєво новим, ніколи раніше невідомим запитом, кешування веде тільки до підвищення перевантаження системи. Пул *Shared* стає компонентом, який пригнічує продуктивність. Часто, але абсолютно невиправдано, для вирішення цієї проблеми багато хто намагається збільшувати розмір пулу *Shared*, що, як правило, тільки погіршує становище.

Коли пул *Shared* неминуче знову заповнюються, він стає ще більшим тягарем, ніж пул меншого розміру, з тієї простої причини, що управління великим повним пулом *Shared* більш трудомістке, ніж управління маленьки м повним пулом.

Єдиним правильним рішенням цієї проблеми є застосування загального *SQL*-коду (тобто повторне використання запитів). Можна також розглянули параметр *CURSOR_SHARING*, який може служити короткотривалим засобом вирішення даної проблеми. Однак єдиний реальний спосіб її вирішення – використання *SQL*-коду багаторазового використання. Навіть в найбільших системах використовується максимум від 10000 до 20000 унікальних застосування *SQL*-коду багаторазового використання. А в більшості систем застосовується лише кілька сотень унікальних запитів.

2.2.4. Пул *Large*

Пул *Large* (Великий) названий так не тому, що являється «великою» структурою (хоча його розмір цілком може бути досить великим), а тому, що він використовується для виділення великих сегментів пам'яті, що перевищують за розміром ті, для обробки яких призначений пул *Shared*.

До введення пулу *Large* в *Oracle 8.0* вся пам'ять виділялася в пулі *Shared*. Це було небажано при використанні функцій, які вимагали «великі» сегменти пам'яті, такі як сегменти пам'яті *UGA* розподіленого сервера. Проблема ще більше ускладнювалася тим, що обробка, яка, як правило, потребувала дуже великого обсягу виділеної пам'яті, повинна була використовувати пам'ять інакше, ніж пул *Shared* керував нею. Пул *Shared* управляє пам'яттю за алгоритмом *LRU,* який прекрасно підходить для кешування і повторного використання даних. Однак при великих сегментах виділення спостерігається тенденція отримання сегмента пам'яті, його використання і припинення роботи з ним – немає ніякої необхідності хешувати вміст цієї пам'яті.

СУБД *Oracle* потребувала механізм, подібному реалізації рециклюючого і утримуючого пулів буферного кеша блоків. Саме такими і є пули *Large* і *Shared* в даний час. Пул *Large* є областю пам'яті, аналогічній рециклюючому пулу, а пул *Shared* більше подібний до утримуючого буферного пулу - якщо виявляється, що що-небудь використовується часто, цей об'єкт зберігається в кеші.

Управління пам'яттю пулу *Large* виконується в купі, багато в чому подібно до того, як *C* управляє пам'яттю за допомогою процедур *malloc ()* і *free ()*. Як тільки ви «звільняєте» сегмент пам'яті, він може бути задіяний іншими процесами. Пул *Shared* не володів ніякими механізмами звільнення сегментів пам'яті. Він повинен був виділяти її, використовувати, а потім припиняти використання. Після закінчення деякого часу, якщо виникала необхідність повторного використання цієї області пам'яті, СУБД *Oracle* повинна була вилучити вміст застарілого сегмента. Проблема застосування тільки пулу *Shared* полягає в тому, що один розмір підходить не для всіх ситуацій.

Пул *Large* використовується перерахованими нижче засобами.

- *Підключеннями за допомогою розподіленого сервера* для виділення області *UGA* в області *SGA*.
- *Паралельним виконанням операторів* для забезпечення виділення буферів повідомлень взаємодії процесів, які використовуються для координації серверів паралельних запитів.
- У деяких випадках *резервним копіюванням* для зберігання буферів дискового введення-виведення процесу *RMAN*.

Як бачите, управління жодною з цих областей пам'яті не повинно виконуватися в буферному пулі типу *LRU*, призначеному для управління невеликими сегментами пам'яті. Наприклад, пам'ять, використовувана підключенням за допомогою розподіленого сервера, ніколи не застосовується повторно після виходу з сеансу, тому вона повинна негайно повертатися в пул. Крім того, як правило, область пам'яті розподіленого сервера досить «велика». Якщо повернутися до раніше наведених прикладів використання параметрів *SORT_AREA_RETAINED*_SIZE або *PGA_AGGREGATE_TARGET*, можна переконатися, що область *UGA* може ставати дуже великою - у всякому разі, її розмір перевищує 4 Кбайт. Розміщення пам'яті багатопоточного сервера в пул *Shared* веде до її розбиття на фрагменти довільних розмірів. Більше того, виявиться, що великі сегменти пам’яті, котрі ніколи не будуть застосуватися повторно, приведуть до старіння і тих сегментів, які могли б використовуватися багаторазово. В результаті база даних буде змушена виконувати додаткову роботу по перебудові цієї структури пам’яті.

Все сказане відноситься і до буферів повідомлень паралельних запитів, оскільки вони непридатні для управління за алгоритмом *LRU*. Вони виділяються і не можуть бути звільненні до тих пір, поки їх використання не буде зупинено. Як тільки вони доставили свої повідомлення, вони більше не і потрібні, і повинні бути негайно звільнені. Це в ще більшій мірі відноситься до резервного копіювання їх використання, вони повинні негайно «зникнути».

Застосування пулу *Large* не обов'язкове при використанні підключень через розподілений сервер, але я рекомендую робити це. При підключенні за

допомогою розподіленого сервера під час відсутності пулу *Large* пам'ять виділяється в пулі *Shared*, як це мало місце в *Oracle 7.3* і попередніх версіях. Це неминуче призведе до зниження продуктивності після закінчення деякого часу і цього слід уникати. Якщо для параметра *DBWR_IO_SLAVES* або *PARALLEL_MAX_SERVERS* встановлено якесь позитивне значення, за замовчуванням пулу *Large* буде присвоєно певний розмір. У разі застосування функціональних засобів, які використовують пул *Large*, його розмір рекомендується встановлювати вручну. Як правило, механізм за замовчуванням не забезпечує вибір значення, оптимального для конкретної ситуації.

2.2.5. Пул *Java*

Пул *Java* був запроваджений у версії *Oracle 8.1.5* для підтримки виконання коду *Java* в базі даних. Якщо процедура, написана на мові *Java, Oracle* використовує цей сегмент пам'яті при обробці такого коду. Параметр *JAVA_POOL_SIZE* служить для фіксації обсягу пам'яті, виділеної пулу *Java* для зберігання всього коду і даних *Java*, властивих окремому сеансу.

Спосіб використання пулу *Java* залежить від режиму роботи сервера *Oracle*. У режимі виділеного сервера пул *Java* включає в себе загальні частини кожного з класів *Java*, які дійсно використовуються в сеансі. В основному це частини, доступні тільки для читання (вектори виконання, методи тощо) і їх розмір становить близько 4-8 Кбайт на один клас.

Таким чином, в режимі виділеного сервера (що, швидше за все, буде мати місце у разі додатків, які використовують процедури, що зберігаються, написані тільки на мові *Java*) загальний обсяг пам'яті, необхідний для пулу *Java*, достатньо помірний і може бути визначений, виходячи з кількості застосовуваних класів *Java*. Слід зазначити, що в режимі виділеного сервера жоден елемент даних, характерний для сеансу, не зберігається в області *SGA*, оскільки ця інформація міститься в області *UGA*, яка в режимі виділеного сервера входить до складу області *PGA*.

При підключенні до *Oracle* через розподілений сервер пул *Java* включає в себе обидва наступних компонента.

- Загальна частина всіх класів *Java*.
- Частина *UGA*, використовувана для зберігання інформації про стан кожного сеансу і виділяється з сегмента *JAVA_POOL* всередині області *SGA*. Інша частина області *UGA* буде, як завжди розміщуватися в пулі *Shared* або в пулі *Large*, якщо він визначений.

Оскільки в *Oracle9i* і попередніх версіях загальний розмір пулу *Java* фіксований, розробникам додатків доведеться оцінити загальний обсяг пам'яті, необхідний їх додаткам, і помножити це значення на число одночасно діючих сеансів, які потрібно підтримувати. Це значення визначить загальний розмір пулу *Java*. Кожен сегмент *Java* області *UGA* буде збільшуватися і зменшуватися в міру необхідності, але слід мати на увазі, що розмір цього пула повинен бути визначений так, щоб в ньому могли одночасно уміщатися всі сегменти *UGA*. В *Oracle10g* і наступних версіях цей параметр може змінюватися, і пул *Java* з часом може збільшуватися і зменшуватися без перезапуску бази даних.

2.2.6. Пул *Streams*

Пул *Streams* (Потоки) - нова структура *SGA*, що з'явилася в *Oracle10g*. Самі потоки - це нова функціональна можливість бази даних, що надається *Oracle9i Release 2* і подальшими версіями. Вона була розроблена в якості інструменту спільного використання/реплікації даних і відповідає заявленому корпорацією *Oracle* курсу на вдосконалення реплікації даних.

Твердження про те, що потоки (*Streams*) «відповідають оголошеному корпорацією *Oracle* курсу на вдосконалення реплікації даних» не означає, що традиційний засіб реплікації *Oracle - Advanced Replication* (Розширена реплікація) – незабаром зникне. Навпаки, воно буде підтримуватися в майбутніх версіях.

Пул *Streams* (який становить до 10 відсотків пулу *Shared*, якщо ніякий пул *Streams* не визначений) служить для буферизації повідомлень черги, використовуваних процесом *Streams* при переміщенні/копіюванні даних з однієї бази даних в іншу. Замість того, щоб застосовувати постійні, що зберігаються на

диску черзі, які викликають перевантаження, процес *Streams* використовує черги в пам'яті. У міру їх заповнення ці черги будуть скидатися на диск. Якщо з якоїсь причини (через програмну помилку, збоїв живлення або з іншої причини) в екземплярі *Oracle*, що використовує чергу в пам'яті, виникає збій, ці черги в пам'яті відновлюються з журналів повторення транзакцій.

Таким чином, пул *Streams* матиме значення тільки в системах, які використовують функціональну можливість *Streams* бази даних. У цих середовищах слід визначити цей пул, щоб уникнути «викрадення» 10% обсягу пулу *Shared* для забезпечення підтримки цієї функціональної можливості.

2.2.7. Автоматичне управління пам'яттю *SGA*

Подібно до того, що існує два способи управління пам'яттю *PGA*, починаючи з *Oracle10g*, існують два способи управління пам'яттю *SGA*: вручну, шляхом установки всіх необхідних параметрів пулу і кеша і автоматично, за допомогою установки лише декількох параметрів пам'яті і єдиного параметра *SGA_TARGET*. Встановлюючи параметр *SGA_TARGET*, ви надаєте екземпляру можливість визначати і змінювати розміри різних компонентів *SGA*.

В *Oracle9i* і попередніх версіях пам'яттю *SGA* можна було управляти тільки вручну - параметр *SGA_TARGET* не існував, а параметр *SGA_MAX_SIZE* представляв граничне значення, а не динамічне цільове значення.

У *Oracle10g* пов'язані з управлінням пам'яттю параметри розбиті на дві групи.

- *Автоматично налаштовуванні параметри SGA*. На даний час ними є *DB_CACHE_SIZE, SHARED_POOL_SIZE, LARGE_POOL_SIZE і JAVA_POOL_ SIZE*.
- *Налаштовуванні вручну параметри SGA*. До них відносяться *LOG_BUFFER, STREAMS_POOL, DB_NK_CACHE_SIZE, DB_KEEP CACHE_SIZE* і *DB_RECYCLE_CACHE_SIZE*.

В *Oracle 10g* в будь-який час можна запросити уявлення *V$SGAINFO* для вияснення того, розміри яких компонентів *SGA* можна змінювати.

Щоб можна було використовувати автоматичне керування пам’яттю *SGA*, в якості значення параметра *STATISTICS_LEVEL* потрібно встановити *TYPICAL* або *ALL*. Якщо збір статистики не включений, база даних не отримає хронологічну інформацію, потрібну для прийняття необхідних рішень щодо розмірів областей пам'яті.

При автоматичному управлінні пам'яттю *SGA* основним параметром завдання розмірів автоматично налаштовуваних компонентів є SGA TARGET, який може динамічно змінюватися під час роботи бази даних аж до значення, заданого параметром *SGA_MAX_SIZE* (яке за замовчуванням встановлюється рівним значенню *SGA_TARGET*; тому, якщо ви плануєте збільшити значення *SGA_TARGET*, перед запуском екземпляра бази даних необхідно збільшити значення *SGA_MAX_SIZE*). База даних буде використовувати обсяг пам'яті, рівний розміру *SGA_TARGET* за вирахуванням розміру будь-яких інших вручну встановлених компонентів, таких як *DB_KEEP_CACHEE_SIZE, DB_RECYCLE_CACHE_SIZE* і інших, і використовує цей обсяг пам'яті для виділення буферного пулу за замовчуванням і пулів *Shared, Large* і *Java*. Під час виконання екземпляр буде динамічно виділяти і перерозподіляти пам'ять між цими чотирма областями пам'яті в міру необхідності. Замість того щоб повертати користувачеві повідомлення про помилку *ORA-04031”Unable to allocate N bytes of shared memory”(ORA-04031* «Неможливо розподілити *N* байт спільно використовуваної пам'яті») у разі нехватки пам'яті в пулі *Shared*. Екземпляр може змншити буферний кеш на певну кількість мегабайт (кратне розміру гранули) і відповідним чином збільшити розмір пулу *Shared*.

Згодом, коли потреба примірника у пам'яті проясниться, розміри рызних компонентів *SGA* стануть більш-менш фіксованими. База даних запам’ятовує також розміри цих чотирьох компонентів під час різних запусків і зупинок бази даних, що позбавляє її від необхідності заново їх визначати при кожному запуску екземпляра. Для цього вона використовує чотири параметри, помічені подвійним символом підкреслення: *DB_CACHE_SIZE, JAVA_POOL_SIZE, LARGE_POOL_SIZE* і *SHARED_POOL_SIZE*. Під час звичайного або термінового зупинення база даних буде записувати ці параметри в файл збережених

параметрів, а потім застосовувати їх під час запуску для установки початкових розмірів кожної з областей.

Крім того, якщо відомо певне мінімальне значення розміру однієї з цих чотирьох областей, цей параметр можна встановити на додаток до параметру *SGA_TARGET*. Екземпляр використовує це значення в якості нижньої межі (найменшого розміру, який може мати дана конкретна область).

2.3. Паралелізм та узгоджене читання

Як було встановлено раніше, однією з ключових проблем при розрозробці багатокористувацьких додатків, що працюють з базою даних, є максимізація паралельного доступу при одночасному забезпеченні можливості кожному користувачеві читати і модифікувати дані в узгодженному вигляді. У цій главі буде детально розглянуто, яким чином *Oracle* досягає *багатоверсійної узгодженості* читання і що це означає для розробника. Також буде предсавлено нове поняття, *узгодженість запису* (*write consistency*) і використовую його для опису роботи *Oracle* не тільки в середовищі читання з узгодженістю прочитаних даних, але також в змішаному середовищі читання/запису.

2.3.1. Що таке управління паралелізмом?

Управління паралелізмом - це колекція функцій, наданих СУБД для організації багатьом людям одночасного доступу до даних і їх модифікації. Як було відзначено раніше, *блокування* – це один з центральних механізмів, за допомогою якого *Oracle* регулює конкурентний доступ для поділу ресурсів бази даних і запобігання «інтерференції» між конкуруючими транзакціями бази даних. Підводячи короткий підсумок, можна сказати, що *Oracle* використовує різноманітність блокувань, включаючи наступні:

- *Блокування ТХ.* Ці блокування встановлюються протягом транзакції, що модифікує дані.

- *Блокування TM і DDL.* Ці блокування гарантують, що структура об'єкта зміниться в процесі модифікації його вмісту (блокування *TM*) або самого об'єкта (блокування *DDL*).
- *Засувки.* Це внутрішні блокування, які використовуються *Oracle* в посередництві доступу до структур даних, які розділяються.

В кожному разі з установкою блокування асоціюються мінімальні накладні витрати. Блокування транзакцій *TX* виключно масштабуються відносно, як продуктивності, так і кардинальності. Блокування *TM* і *DDL* застосовуються в найменш обмежуваному режимі, наскільки це можливо. Засувки і черги дуже легші і швидші (черги трохи важче з цих двох, оскільки їх можливості більші). Проблеми виникають через погано спроектованих додатків, які утримують блокування довше, ніж це необхідно і викликають блокування бази даних. Якщо ви ретельно проектуєте свій код, механізми блокування *Oracle* дозволять будувати масштабовані додатки з високим ступенем паралелізму.

Однак підтримка паралелізму СУБД *Oracle* не обмежується ефективним блокуванням. Вона реалізує *багатоверсійну* архітектуру, яка забезпечує контрольований, але при цьому надзвичайно паралельний доступ до даних. Багатоверсійність характеризує здатність *Oracle* паралельно матеріалізувати безліч версій даних і є механізмом, за допомогою якого *Oracle* забезпечує узгоджене читання і подання даних (тобто узгоджені результати на певний момент часу). Досить приємний ефект від багатоверсійності полягає в тому, що читач даних ніколи не блокується письменником даних. У цьому полягає одна з фундаментальних відмінностей *Oracle* від інших баз даних. Запит, який тільки читає інформацію, в *Oracle* ніколи не блокується, він ніколи не вступає у взаємоблокування (*deadlock*) з іншим сеансом і ніколи не видає відповідь, яка відсутня в базі даних.

Існує короткий період під час обробки розподілених *2PC*, коли *Oracle* запобігає доступ до інформації з читання. Оскільки ця обробка трапляється рідко і скоріше є винятком (проблема стосується тільки запитів, що стартують між фазами підготовки та фіксації і намагаються прочитати дані перед тим, як

приходить команда зафіксувати транзакцію). Тому не потрібно поглиблюватись в цю тему.

Багатоверсійна модель *Oracle* для узгодженості читання за замовчуванням застосовується на *рівні оператора* (для кожного запиту), але також може застосовуватися на рівні транзакції. Це означає, що кожен оператор *SQL*, відправлений в базу даних, бачить узгоджене з читання подання бази, і якщо ви хочете отримати узгоджений з читання стан бази на рівні транзакції (безлічі *SQL*-операторів), то може це зробити.

Основне призначення транзакції - переводити базу з одного узгодженого стану в інший. Стандарт *ISO* специфікує різні *рівні ізоляції транзакцій*, які визначають, наскільки «чутлива» одна транзакція до змін, зробленими іншою транзакцією. Чим вище рівень чутливості, тим вища ступінь ізоляції, яку повинна забезпечувати база між транзакціями виконуваними вашим додатком.

2.3.2. Рівні ізоляції транзакції

Стандарт *ANSI/ISO* визначає чотири рівні ізоляції транзакцій, з різними можливими наслідками для однакового сценарію транзакцій. Тобто, одна і та ж робота, виконана в одній і тій же манері, з одним і тим же введенням може призводити до різних відповідей - в залежності від обраного рівня ізоляції. Ці рівні ізоляції визначені в термінах трьох «феноменів», які або дозволені, або немає на заданому рівні ізоляції.

- *Брудне читання*. Сенс цього терміну так само поганий, як і він звучить. Вам дозволено читати незафіксовані або брудні (*dirty*) дані. Ви досягаєте такого ефекту, відкриваючи файл операційної системи, який хтось інший записує або читає, незалежно від того, які дані там виявляться. Цілісність даних не гарантується, зовнішні ключі порушуються, обмеження унікальності ігноруються.

- *Невідтворюваність читання.* Це просто означає, що якщо ви читаєте рядок в момент часу *T1*, а потім намагаєтеся перечитати її в момент часу *T2*, то рядок може за цей час змінитися. Він може зникнути, змінитися і так далі.

- *Фантомне читання.* Це означає, що якщо ви виконаєте запит в момент *T1*, а потім повторите його в момент *T2*, то в базі можуть з'явитися додаткові рядки, що може вплинути на отримані результати. Відмінність фантомного читання від невідтворюваного читання полягає в тому, що дані, які вже прочитані, не змінюються, але вашому критерію запиту задовольняє більше даних, ніж раніше.

Рівні ізоляції *SQL* визначені на основі того, допускають вони чи ні кожен з описаних феноменів. Я вважаю, цікаво відзначити, що стандарт *SQL* не вимагає специфічної схеми блокування або певної поведінки, а скоріше описує рівні ізоляції в термінах цих феноменів, допускаючи існування різноманітних механізмів блокування/паралелізму (див. табл. 2.1).

Таблиця 2.1. Рівні ізоляції *ANSI*

Рівень ізоляції	**Брудне читання**	**Невідтворюваність читання**	**Фантомне читання**
READ UNCOMMITTED	Допускається	Допускається	Допускається
READ COMMITTED		Допускається	Допускається
REPEATABLE READ			Допускається
SERIALIZABLE			

СУБД *Oracle* явно підтримує рівні ізоляції *READ COMMITTED* і *SERIALIZABLE*, як вони визначені в стандарті. Однак, це не все. Стандарт *SQL* намагався встановити рівні ізоляції, які допускають різні ступені узгодженості запитів, які виконуються на кожному рівні. *REPEATABLE READ* - це рівень ізоляції, на якому стандарт *SQL* вимагає гарантії узгодженості читання по запиту. У визначенні стандарту *SQL READ COMMITTED* не дає узгоджених результатів,

а *READ UNCOMMITTED* - це рівень, який використовується для не блокуючого читання.

Однак в *Oracle READ COMMITTED* володіють усіма атрибутами, необхідними для забезпечення узгоджених з читання запитів. В інших базах даних запити *READ COMMITTED* можуть і будуть повертати відповіді, які ніколи не існували в базі в певний момент часу. Більше того, *Oracle* також підтримує дух *READ UNCOMMITTED*. Мета брудного читання - забезпечити можливість не блокуючого читання, коли запит не блокується і не блокує поновлення деяких даних. Однак *Oracle* не потребує брудного читання для досягнення цієї мети, і не підтримує його. Брудне читання змушені реалізовувати інші бази даних, щоб забезпечити можливість не блокуючого читання.

На додаток до перерахованих чотирьох рівнів ізоляції *ISO*, *Oracle* пропонує ще один рівень - *READ ONLY*. Транзакція *READ ONLY* - еквівалент транзакції *REPEATABLE READ* або *SERIALIZABLE*, які не виконують ніяких модифікацій в *SQL*. Транзакція, яка використовує рівень ізоляції *READ ONLY*, бачить тільки ті зміни, які були зафіксовані на момент початку цієї транзакції, а вставки, оновлення та видалення в цьому режимі не допускаються (інші сеанси можуть оновлювати дані, але не транзакція *READ ONLY*). Використовуючи цей режим, ви можете досягти рівнів ізоляції *REPEATABLE READ* і *SERIALIZABLE*.

Тепер давайте поговоримо про те, як саме багатоверсійність і узгодженість читання укладається в схему ізоляції, і як бази даних, які не підтримують багатоверсійність, досягають того ж результату. Ця інформація буде корисна тим, хто використовує інші бази даних, і вірить, що розуміє, як повинні працювати рівні ізоляції. Цікаво також бачити, як стандарт, призначений для усунення відмінностей між базами даних - *ANSI/ISO SQL* - в дійсності справляється з цим. Стандарт, хоча і дуже деталізований, може бути реалізований дуже різними способами.

2.3.2.1. *READ UNCOMMITTED*

Рівень ізоляції *READ UNCOMMITTED* допускає брудні читання. *Oracle* не використовує брудне читання і не дозволяє його застосовувати. Основна мета

рівня ізоляції *READ UNCOMMITTED* в тому, щоб забезпечити засноване на стандартах визначення, відповідне для не блокуючого читання. Як ми бачили, Oracle забезпечує не блокуюче читання за замовчуванням. Ви будете змушені виконувати блокування запиту *SELECT* в базі даних. Кожен окремий запит, будь то *SELECT, INSERT, UPDATE, MERGE* або *DELETE*, виповнюється в узгодженій читанню манері. Може здатися забавним називати оператор *UPDATE* запитом, але так і є. Оператори *UPDATE* складаються з двох компонентів: компонент читання визначається конструкцією *WHERE,* а компонент запису - конструкцією *SET*. Оператори *UPDATE* читають і пишуть інформацію в базу даних, як це роблять всі оператори *DML*. Спеціальний випадок однорядкового *INSERT* з використанням конструкції *VALUES* - єдиний виняток з цього, оскільки цей оператор не має компонента читання, а тільки компонент запису.

2.3.2.2. *READ COMMITTED*

Рівень ізоляції *READ COMMITTED* встановлює, що транзакція може читати тільки ті дані, які були зафіксовані в базі даних. Немає ніякого брудного читання. Можуть бути не відтворювані читання (тобто повторні читання того ж рядка можуть повертати різні відповіді в одній і тій же транзакції) і фантомні читання (тобто знову вставлені і зафіксовані рядки стають видимими запиту, хоча не були видимі раніше в тій же транзакції). *READ COMMITTED* - можливо, найбільш часто вживаний рівень ізоляції у всіх додатках баз даних і він є режимом за замовчуванням для баз даних *Oracle*. Застосування інших рівнів ізоляції можна побачити рідко.

Однак забезпечення рівня ізоляції *READ COMMITTED* не так просто, як здається. Очевидно, з урахуванням вище викладених правил, запит, виконуваний в будь-якій базі даних з використанням рівня ізоляції *READ COMMITTED*, поводиться однаково, чи не так? А ось і ні. Якщо запитати безліч рядків одним оператором, то майже в будь-якій іншій базі даних ізоляція *READ COMMITTED* буде настільки ж погана, як і брудне читання, залежно від реалізації.

В *Oracle*, яка використовує багатоверсійність і узгоджені з читання запити, відповідь на запит до *ACCOUNTS* буде однакова, як в прикладі з *READ*

COMMITTED, так і з *READ UNCOMMITTED*. *Oracle* реконструює модифіковані дані за їх станом на момент запуску запиту, повертаючи відповідь, яка містилася в базі в той момент, коли був виданий запит.

2.3.2.3. *REPEATABLE READ*

Метою *REPEATABLE READ* є забезпечення такого рівня ізоляції, який дає узгоджені коректні відповіді і запобігає втраті оновлень.

Втрачені поновлення: ще одна проблема переносимості.

Поширене використання *REPEATABLE READ* в базах даних, які *застосовують розділяєме блокування читання*, призначене для запобігання втрачених оновлень. Якщо ми маємо рівень ізоляції *REPEATABLE READ* в базі даних, яка застосовує розділяєме блокування читання (а не багатоверсійність), то помилки втрачених оновлень відбутися не можуть. Причина втрачених оновлень не виникає в цих базах даних тому, що проста вибірка даних залишає на них блокування, і дані, одного разу прочитані в нашій транзакції, не можуть бути модифіковані ніякий іншою транзакцією. Тепер, якщо ваша програма передбачає, що *REPEATABLE READ* має на увазі, що "втрачені поновлення неможливі", то ви зіткнетеся з болючим сюрпризом при перенесенні вашого додатку в базу даних, що не використовує поділюваних блокувань читання як лежить в основі механізму управління паралелізмом. Хоча це і звучить добре, ви повинні пам'ятати, що якщо залишати установлені колективні блокування читання на всіх прочитаних даних, то це, безумовно, обмежить можливості паралельного читання і модифікації. Таким чином, хоча такий рівень ізоляції в цих базах даних покликаний запобігати втрачені поновлення, це досягається повним виключенням можливості паралельного виконання операції!

2.3.2.4 *SERIALIZABLE*

Зазвичай цей вважається найбільш обмежуючим рівнем ізоляції транзакцій, але він забезпечує максимальну ступінь ізоляції. *SERIALIZABLE* працює в середовищі, в якій можна уявити, що жоден інший користувач не

модифікує інформацію в базі даних. Будь-який прочитаний рядок залишається незмінним до повторного прочитання, і будь-які запити, які виконуються гарантовано і повертають одні і ті ж результати протягом існування транзакції. Наприклад, якщо ми виконаємо запит:

- *Select * from T;*
- *Begin dbms lock.sleep (60 * 60 * 24); end;*
- *Select * from T;*

то відповідь, отриманий з *T*, буде однаковим, навіть незважаючи на те, що між спробами його виконання ми проспали 24 години (або ми отримаємо помилку *ORA-1555: snapshot too old ("ORA-1555:* застарілий знімок*")*. Цей рівень ізоляції гарантує, що два однакових запиту завжди повернуть однакові результати. Побічні ефекти (зміни) від інших транзакцій залишаються невидимими для запиту, незалежно від часу його виконання.

В *Oracle* транзакція *SERIALIZABLE* реалізована так, що узгодженість читання, яка зазвичай забезпечується на рівні оператора, поширюється на всю транзакцію.

Як зазначалося раніше, в *Oracle* також є рівень ізоляції, позначений як *READ ONLY*. Він володіє всіма якостями рівня ізоляції *SERIALIZABLE*, АЛЕ забороняє модифікацію. Слід зазначити, що користувач *SYS* (або користувач, підключений як *SYSDBA*), не може мати транзакцій *READ ONLY* або *SERIALIZABLE*. В цьому відношенні користувач *SYS* - особливий.

Замість забезпечення узгодженості результату на момент запуску оператора, це забезпечується на момент початку транзакції. Іншими словами, СУБД *Oracle* використовує сегменти відкату для реконструкції даних в тому вигляді, в якому вони існували на початку транзакції, а не під час запуску оператора.

Це дуже глибока думка: база даних вже знає відповідь на будь-яке питання, який ви можете задати, перш ніж ви його задаєте.

Цей рівень ізоляії обходить не дарма, і ціною є можливість помилки:

- *ERROR at 1ine 1:*

- *ORA-08177: can not serialize access for this transaction*
- ПОМИЛКА в рядку 1:
- *ORA-08177*: неможливо серіалізовать доступ для цієї транзакції

Ви отримуєте це повідомлення всякий раз, коли намагаєтеся оновити рядок, змінений з моменту початку транзакції.

Oracle намагається зробити це просто на рівні рядка, але ви можете отримати ошіб- у *ORA*-08177, навіть коли рядок, яку ви хочете модифікувати, не була змінена. Помилка *ORA*-08177 може траплятися через те, що деяка рядок (рядки) були моди- ваних в тому ж блоці, де знаходиться цікавить вас рядок.

СУБД *Oracle* дотримується оптимістичного підходу до сериалізації - вона робить ставку на той факт, що дані, які ваша транзакція збирається оновити, не будуть оновлені ніякиою іншою транзакцією. Зазвичай так і трапляється, і ставка виграє, особливо в системах типу *OLTP* з швидкими транзакціями. Якщо ніхто інший не оновлює дані під час вашої транзакції, цей рівень ізоляції, зазвичай знижує ступінь паралелізму в інших системах, забезпечає тут ту ж ступінь паралелізму, що і без транзакцій *SERIALIZABLE*. Недоліком є те, що існує ймовірність отримати помилку *ORA-08177*, якщо ставка не виправдовується. Однак якщо подумати, це того варте. Використовуючи транзакцію *SERIALIZABLE*, напевно ви не повинні чекати, що та ж інформація буде оновлена інший транзакцією. Якщо ж таке можливо, ви повинні використовувати *SELECT ... FOR UPDATE*, і це дозволить сериалізувати доступ. Таким чином, застосування рівня ізоляції *SERIALIZABLE* досяжно і ефективно, якщо дотримуються такі умови.

- Велика ймовірність, що ніхто інший не буде модифікувати одні й ті ж дані.
- Потрібна согласованість читання уровня транзакції.
- Транзакціі будуть короткими.

У *Oracle* вважають такий метод досить масштабованим, щоб запускати все їх *TPC-C* (промисловий стандарт *OLTP*).

В багатьох інших реалізаціях ви виявите, що ті ж цілі досягаються застосуванням розділюваних блокувань читання з властивими їм

взаємоблокуваннями і тому подібним. В *Oracle* немає необхідності встановлювати блокування, але якщо в інших сеансах відбувається зміна даних, які ви збираєтеся змінити, то видається помилка *ORA-08177*. Однак ця помилка виникає не так часто, як взаємоблокування в інших системах.

Але - завжди є якесь "але". Ви повинні подбати про те, щоб чітко зрозуміти різницю між рівнями ізоляції і їх реалізаціями. Пам'ятайте, що при рівні ізоляції *SERIALIZABLE* ви не побачите ніяких змін, внесених в базу даних після запуску вашої транзакції - до тих пір, поки ви її не зафіксуєте.

Висновки до розділу

У цьому розділі було розглянуто структуру пам'яті *Oracle.* Розпочавши з рівня процесів і сеансів, дослідивши при цьому області *PGA* і *UGA* і їх взаємозв'язок. Було з’ясовано, який вплив режим підключення до *Oracle* буде надавати на організацію пам'яті. Підключення через виділений сервер передбачає більш інтенсивне використання пам'яті процесом сервера, ніж підключення з засобом розподіленого сервера, але підключення через розподілений сервер потребує значно більшого обсягу *SGA.* Потім було розглянуто основні структури самої області *SGA.* Та з’ясовано відмінності між пулами *Shared* і *Large*, і було встановлено причини, з яких нам може бути потрібен пул *Large* для «економії» використовуваного обсягу пулу *Shared*. Та ознайомлення з пулом *Java* і специфікою його застосування в різних ситуаціях, а також з буферним кешем блоків і можливістю його поділу на менші, більш спеціалізовані пули.

Також було розібрано рівні ізоляції транзакції, для чого вони потрібні і які проблеми можна вирішувати, через використання кожного з рівнів.

РОЗДІЛ 3
ПРОЕКТУВАННЯ ЕФЕКТИВНОЇ СХЕМИ. РЕКОМЕНДАЦІЇ ПРОГРАМУВАННЯ НА *PL/SQL*

Робота програми багато в чому залежить від його фізичної реалізації. При використанні невірної структури даних в першу чергу страждає продуктивність. У цій главі буде розглянуто ряд моментів, які необхідно враховувати при проектуванні схеми.

3.1. Базові принципи проектування схеми

У цьому розділі розглядаються базові методи, які слід прийняти на озброєння при проектуванні схеми. Буде вивчена перевірка цілісності і оптимізація запитів.

3.1.1. Застосування цілісності даних

З особистого опиту пам’ятаю історію про консультанта, який хотів видалити всю довідкову цілісність з бази даних і зробити це в додатку. Я пояснив, що це не є хорошою ідеєю з наступних причин:

- Крім цього додатка, існують і інші, бажаючі працювати з цим даними. Якщо керуючі цілісністю даних правила заховані в клієнтському додатку, вони не будуть викликатися постійно і змінити їх буде досить складно.
- Управління цілісністю даних на стороні клієнта уповільнює роботу, оскільки вимагає додаткових звернень до сервера.
- Управління цілісністю на стороні клієнта вимагає написання значно більшого обсягу коду.

На мою думку, розробники прагнуть до виконання подібних операцій на клієнті через проходження філософії «адміністратори бази даних проти розробників», а не «адміністратори бази даних і розробники працюють разом». Якщо управління цілісністю даних здійснюється в базі даних, то розробники вважають, що вони втрачають контроль над процесами. Якщо ж це робиться в додатку, розробники стають господарями становища. Однак подібні міркування недалекоглядні. Ні розробники, ні адміністратори бази даних не володіють даними. Дані є власністю третьої особи - кінцевого користувача. І зовсім не в інтересах кінцевого користувача ховати бізнес-правила в клієнтську програму.

3.1.2. Необхідність у збереженні цілісності посилань в базі даних

Правила, що стосуються цілісності даних, безсумнівно, повинні знаходитися в базі даних, що гарантує їх узгоджену реалізацію і застосування. Обумовлені додатком правила, тобто правила, котрі стосуються даних тільки з точки зору використання їх програмою, можуть розміщуватися безпосередньо в додатку. Нижче наведені причини, за якими цілісність даних повинна реалізовуватися по можливості базою даних:

- **Неможливість використання деяких засобів** (таких, як перезапис запиту). Якщо база даних нічого не знає про зв’язки між об'єктами, ці кошти стаюжть марними. Виключення становлять випадки, коли правила, що застосовуються до даних, докладно пояснюються
- **В деякий момент часу може виникнути небезпека для цілісності даних**. Фактично в кожній системі, в якій обмеження зовнішнього ключа застосовуються за межами бази даних, з'являються дочірні рядки, які залишилися без батьківських рядків. У такій системі досить запустити запит:

➢ *select foreign_key_columns from child_table MINUS select primary_key columns from parent.*

Результат може бути дивним! При перевірці даних ви можете виявити значення *NULL* там, де їх не повинно бути, і дані, які не погоджуються з встановленими бізнес-правилами (наприклад, дані за межами діапазону). Це відбувається через неузгодженість програми та бізнес правил.

- **Цілісність, яка забезпечується сервером, працює швидше.** Безсумнівно, при повній відсутності цілісності даних, система працює швидше, оскільки перевірка цілісності вимагає додаткового тільних дій. Але на сервері це працює швидше, чим самостійно написана користувачем програма, і узгоджено виконується незалежно від додатку.
- **База даних надає більше інформації.** При збереженні правил цілісності в базі даних система є більш документованою. Виконавши простий запит, можна дізнатися, що з чим і як зв’язано.

І в якості останньої причини необхідності використання цілісності даних, яка забезпечується сервером, давайте подивимося, що може статися при спробі самостійного виконання цієї роботи.

3.1.3. Проведення перевірки цілісності на стороні клієнта

Існують ситуації, в яких корисно виконувати перевірку цілісності на стороні клієнта. Але важливо пам'ятати, що в кінцевому рахунку це *всеодно має виконуватися в базі даних.* Перевірка цілісності на стороні клієнта не буде заміною перевірки цілісності на сервері, а послужить додатковим компонентом. Нижче наводяться деякі причини застосування цілісності на клієнті:

- **Досвід кінцевого користувача.** Під час введення даних користувача можуть виявити помилку. Таким чином, скорочується імовірність надходження в базу невірних даних, а для користувачів немає необхідності чекати вибору команди *Save.*
- **Скорочення споживання ресурсів сервера.** В сервера не вникає необхідності виконувати непотрібну роботу, якщо на клієнті були помічені і виправлені невірні дані.

Але крім позитивних моментів, є й негативні. Головною незручністю організації перевірки цілісності на клієнті являється наявність двох місць з правилами. Якщо з плином часу правила змінюються, то необхідно переконатися в тому, що зміни відбулися в обох місцях. Клієнт не зможе ввести інформацію в базу даних, якщо додаток буде працювати за старими правилами. Це рівнозначно

тому, як якби користувач працював з набором даних і був абсолютно впевнений в їх достовірності, але при передачі їх з програми в базу даних він раптом виявив би, що база даних не приймає ці дані. Грамотне управління конфігурацією і правильні принципи створення програмного забезпечення зменшать ймовірність виникнення подібної ситуації.

3.1.4. Використання адекватного типу даних

Нерідко можна зустріти системи, в яких:

- Для зберігання дати та часу застосовується рядок.
- Для зберігання чисел застосовується рядок.
- Для зберігання всіх рядків використовується *VARCHAR2(4000).*
- Для зберігання всіх рядків використовується *CHAR(2000)*, при цьому витрачається величезна кількість місця і доводиться багато разів викликати функцію *trim*.
- Для розміщення тексту застосовується тип *BLOB.*

Я завжди застосовую дуже просте правило: для дат використовувати тип даних - дати, для чисел - числа, а для рядків - рядки. Не варто застосовувати тип даних для зберігання чогось ще, крім того, для чого він призначений. Потрібно по можливості використовувати найбільш відповідний тип. До того ж дати можна порівнювати тільки з датами, рядки з рядками, а числа з числами. Якщо дати і числа зберігаються в рядку або використовується невідповідна довжина, то система терпить збитки:

- Втрачається можливість редагування даних при виконанні вставки в базу даних, при перевірці відповідності дати поточній даті і правильності чисел.
- Знижується продуктивність.
- Потенційно зростає потреба в пам'яті.
- Зменшується цілісність даних.

В системах, де числа і дати зберігаються в рядках, зазвичай виникають помилки *ORA-01722* (невірне число) і *ORA-01858* (виявлений нечисловий символ там, де очікувалося число).

3.1.5. Зменшення цілісності даних

З багатьох причин не слід застосовувати невідповідний тип даних, але першою і найголовнішою з них є цілісність даних. У системах, що використовують рядки для дат або чисел, будуть міститися деякі записи з датами, які не є датами, і з числами, які не є числами. Якщо дозволити застосовувати будь-які символи в полях дат, то через якийсь час в цих полях обов'язково будуть «брудні» дати.

Без застосування правил цілісність даних знаходиться під питанням. Припустимо, що необхідно написати функцію, яка конвертує рядки в дати, але повертає *NULL* при неможливості перетворення. Якщо є дата 01/02/03, то чи зрозуміло, яка це дата? Ця дата представлена у вигляді рр/мм/дд, дд/мм/рр чи як-небудь ще?

3.1.6. Зростання потреби в просторі для зберігання даних

Крім використання адекватних типів даних (число, дата або рядок), слід визначати тип, як можна точніше. Наприклад, для полів з кількістю символів менше 30 слід застосовувати *VARCHAR2(30)*, а не *VARCHAR2(4000).*

Розробник моделей, який входить в мою групу, віддає перевагу визначати всі поля як *VARCHAR2* максимальної довжини. Але це означає, що таблиця з 20 полями *VARCHAR2* матиме розмір від 2000 до 4000 байт. Я намагався йому пояснити, що ми стараємося визначати дані з відповідною довжиною і іменами, щоб розуміти, що є у нас в базі даних. Він відповів мені, що це не дає витрат, оскільки *Oracle* просто зберігає довжину і т.д. Я сумніваюся в його словах, але ніяк не можу знайти документ, в якому говорилося б про внутрішні принципи зберігання інформації в *Oracle.*

Я відповів йому так: «Це повинен був зробити розробник моделей!» Розробники моделей зобов'язані знати про те, що життєво важливо використовувати відповідну довжину при визначенні полів! Давайте на хвилину забудемо про таку річ, як зберігання інформації, і задамо наступні питання:

- Що може статися при роботі користувачів з інструментом запиту, який задає формат кожному полю на підставі ширини стовпчика в базі даних? Для

того щоб побачити другий, третій і т.д. стовпці, користувачам доведеться застосовувати смугу прокрутки.

- Припустимо, що деякий код готує запит, який обирає десять стовпців з типом даних *VARCHAR2*. Для покращення продуктивності розробники вважають за краще виконувати вибірку масиву (дуже важливо), що складається, скажімо, з 100 рядків (дуже типово). Таким чином, для цього завдання розробникам потрібно 4000 x 10 x 100 = близько 4 Мбайт оперативної пам'яті! А якщо буде 10 *полів VARCHAR2(80)*? Це складе всього близько 78 Кбайт. Поцікавтеся у розробника моделей, скільки оперативної пам'яті він готовий віддати цій системі.
- Припустимо, що розробники почали створювати форму введення інформації для розміщення даних в базі. Це поле може бути довжиною в 4000 символів і перше ім'я може мати довжину в 4000 символів. Як дізнатися, які дані будуть вводитися насправді?

Розробник моделі даних повинен розглядати довжину поля в якості обмеження. Точно так само, як при використанні первинного і зовнішнього ключів він застосовує відповідну довжину полів. Завжди можна збільшити довжину поля за допомогою команди :

- *alter table t modify* з *varchar2 (bigger_number).*

Але не існує причин, по яким варто було б ставити максимальну довжину поля. Це буде шкодити роботі додатків, оскільки вони будуть помилково займати велику кількість оперативної пам'яті. Тільки подумайте про вибірку масиву з точки зору сервера додатків. Тепер йтиметься не просто про 4 Мбайт, а про 4 Мбайт, помножених на число підключень. І якщо в даному прикладі говорилося про деяку кількість пам'яті для *єдиного запиту*, то в дійсності буде виконуватися одночасно безліч запитів.

Тип *CHAR(2000)* буде поглинати 2000 байт пам'яті незалежно від того, чи розміщується символ *a*, рядок *«hello world»* або ж 2000 символів. Тип даних *CHAR* завжди доповнюється пробілами. Крім того, якщо виникає необхідність у використанні індексу в системі, то слід остерігатися складнощів, пов'язаних з пам'яттю. Розглянемо наступний приклад:

- *create table t (a varchar2 (4000), b varchar2 (4000);*

  Table created.
- *create index t_idx on t (a);*

  ERROR at line 1:

  ORA-01450: maximus key length (3218) exceeded

В *Oracle9i* максимальна довжина ключа більша, але все одно існують обмеження. Наприклад, індекс по *T(a, b)* в *Oracle9i* з розміром блоків 8 Кбайт викличе помилку *ORA-01450:* довжина ключа (6398) виходить за допустимі межі.

У моїй системі розмір блоків дорівнює 8 Кбайт. Мені необхідно використати щонайменше блоки в 16 Кбайт для індексування *єдиного* стовпчика. Але навіть якщо я спробую створити пов'язаний індекс по *T(A, B)*, то можу потерпіти невдачу.

Те ж саме справедливо для числового типу і *TIMESTAMP*, нового типу даних в *Oracle9i:* для кращого визначення цілісності даних і для надання додатком більшої кількості інформації про дані, слід належним чином задавати масштаб і точність для цих полів.

Не варто піддаватися на вмовляння використовувати будь-який інший тип даних, а не *DATE* або *TIMESTAMP* для дат, або ж застосовувати *VARCHAR2* для чисел. Слід ставити відповідний і коректний тип даних, що дає гарантію отримання максимальної продуктивності і, що понад важливо, захисту цілісності даних.

3.2. Ефективне програмування на PL/SQL

Мова *PL/SQL* є мовою програмування *3GL* і процедурних розширенням до *SQL* в *Oracle*. Вперше він був представлений у версії 6 і дозволяв кодувати «анонімні блоки» в клієнтських додатках, а також розміщувати їх в базі даних для обробки. У *Oracle 6* були відсутні збережені процедури, пакети і тригери. Можливість зберігання *PL/SQL* прийшла в базу даних у 1992 році з версією 7.

На сьогоднішній день *PL/SQL* є правомочним, зрілим, повнофункціональною мовою програмування *3GL*. Я вважаю, що *PL/SQL* незаслужено рідко застосовується в додатках *Oracle*, в результаті чого не вдається повністю розкрити його потенціал. Про *PL/SQL* часто забувають, вважаючи його «застарілою» мовою. Його уникають застосовувати в якості «сучасної» мови (вибираючи *Java* або будь-яку іншу).

Тут буде показано, чому необхідно використовувати *PL/SQL* при роботі з додатками. Крім того, я зупинюся на деяких найбільш важливих методиках програмування, які забезпечують ефективність і високу продуктивність при роботі з програмами на *PL/SQL*.

3.2.1. Вибір *PL/SQL*

Питання, яке краще обговорити на початку, звучить так: «Чому ви вважаєте за краще використовувати на саме *PL/SQL*, незважаючи на те, що *Oracle* підтримує Java, *PL/SQL* і *C* в базі даних?»

На користь можливостей *PL/SQL* вельми красномовно говорить те, що на цій мові написані цілі продукти. Ось декілька прикладів:

- Розширена реплікація *Oracle* була повністю реалізована на *PL/SQL* в версії бази даних 7.1.6. Згодом частина її була переписана на *C*, оскільки ця мова забезпечує кращу продуктивність. Але основна частина функцій реплікації залишилася на *PL/SQL*.
- Комплект додатків *Oracle* (Управління Персоналом (*HR*), Фінанси, Планування ресурсів підприємства (*ERP*) і т.д.) були написані на *PL/SQL*. Сьогодні для обробки графічного інтерфейсу користувача проміжного програмного забезпечення також застосовується *Java*, але спочатку весь комплект був написаний на *PL/SQL*. В даний час майже всі дані, як і раніше опрацьовуються з його допомогою.
- Движок (*Workflow*) *Oracle*, який живе в базі даних, написаний на *PL/SQL*.

- Адміністративний інтерфейс для бази даних реалізований на *PL/SQL*. Він включає в себе *DBMS_STATS, UTL_OUTLN, DBMS_RESOURCE_MANAGER* і т.д.

Багато розробників ігнорують цю мову, не дивлячись на його доведену потужність і багатогранність. Вони наполегливо намагаються реалізувати ту чи іншу функціональність за допомогою інших мов, хоча *PL/SQL* могла б допомогти у вирішенні цього завдання і зробити це набагато швидше і ефективніше. Все, що необхідно для застосування *PL/SQL*, це наявність гарного текстового редактора і *SQL*Plus*. Не так вже й складно почати роботу з *PL/SQL*.

3.2.2. *PL/SQL* - найефективніша мова обробки даних

При роботі з даними *PL/SQL* є однією з найбільш простих і продуктивних мов з точки зору застосування. Розглянемо просту програму, в якій виконуються вибірка з бази даних і деяка обробка цієї вибірки. Програма *PL/SQL* буде виглядати так:

- *Create or replace procedure process_data (p_inputs in varchar2)*
- *As*
- *Begin*
- *For X in (select * from emp where ename like p_inputs)*
- *Loop*
- *Process (X);*
- *End loop*
- *End;*

Необхідно звернути увагу на наступні моменти:

- Типи даних *PL/SQL* є типами даних *SQL*. Між типами мови програмування і базою даних не потрібно ніяких перетворень - вони однакові
- Мови *SQL* і *PL/SQL* легко доповнюють одна одну і можуть застосовуватися разом, так що існує тісний взаємозв'язок між мовою і базою даних. Розглянемо рядок *For X in (select ...)*. Компілятор знає, як визначити *X*,

неявно створює тип запису і поміщає дані, отримані в результаті запиту, до цього запису.

- Немає необхідності у відкритті та закритті запиту.
- Користувач захищений від багатьох змін, вироблених в базі них. Якщо в таблиці необхідно видалити або додати стовпець, ця процедура змінюватися не буде. Для цього коду виправлення не будуть потрібні.

Крім цього, *PL/SOL* виконує неявне кешування курсору. Якщо 1000 разів викликати процедуру і використовувати різні вхідні дані, то можна виявити, що розбір оператора *SELECT* здійснювався лише одного разу, в той час, як сама процедура була виконана 1000 разів. Принцип «*розберемо один раз, а виконаємо багаторазово*» закладений безпосередньо в мову, так що *PL/SOL* розуміє всю його важливість для продуктивності і масштабованості.

Тепер порівняємо цю програму з еквівалентним кодом на *Java/JDBC:*

```
Static PreparedStatement pstmt= null;
public static void process_data (Connection conn, String inputs)
throws Exception
{
int  empno;
String ename;
String job;
int mgr;
Srting hiredate;
Int sal;
int comn;
int deptno;
    if (pstmt == null)
            pstmt = conn.prepareStatement
            ( "select * from emp where ename like ? ");
    pstmt.setString (1, inputs);
```

```
        ResultSet rset = pstmt.executeQuery ();
        while (rset.next () )
        {
                empno      = rset.getInt (1);
                ename      = rset.getstring (2);
                job        = rset.getstring (3);
                mgr        = rset.getInt (4);
                hiredate   = rset.getSrting (5);
                sal        = rset.getInt (6);
                comm       = rset.getint (7);
                deptno     = rset.getInt (8);
                // process   (empno, ename, job, mgr, hiredate,
                // sal, comn, deptno);
        }
        rset.close ();
}
```

Жоден пункт з попереднього списку не виконується:

- Типи *Java* не відповідають типам *SQL*, і необхідно дуже уважно і обережно вибирати типи *Java*. У *Oracle* стовпець з типом даних *NUMBER* може містити 38 знаків. Який тип *Java* слід використовувати, щоб витягти стовпець з типом даних *NUMBER* і при цьому не втратити точність або не переповнити його?
- Не існує ув'язки між мовою *SQL* і *Java/JDBC*. Необхідно всіма діями керувати вручну, оскільки *Java* є повністю процедурною *API.*
- Користувач не захищений від змін в базі даних. Якщо здійснюється вставка, перетворення або видалення стовпця, необхідно також змінювати і код *Java*.
- Слід виконувати кешування операторів вручну (в прикладі це *PreparedStatement, pstmt*).

Я ні в якому разі не збираюся критикувати *Java* або *C* (*C* є однією з моїх улюблених мов програмування). Відповідний код на *C* настільки ж багатослівний і володіє тими ж недоліками, що і *Java* в даному прикладі. Ці мови, безумовно, прекрасні. Але якщо для обробки даних необхідно виконати великий обсяг коду на *SQL* і зовсім невелику кількість процедурної роботи, то краще застосувати *PL/SQL*, оскільки вона є найбільш продуктивною мовою з точки зору і швидкості виконання, і ефективності роботи розробника.

3.2.3. Можливості перенесення і повторного використання *PL/SQL*

Чи можна стверджувати, що *PL/SQL* є більш переносними і повторно використовуваною, ніж будь-які інші мови? Оскільки це твердження зазвичай викликає подив і недовіру, я поясню свою точку зору.

Безумовно і *Java,* і *C* також є переносними, і повторно використовуваними. Однак Java є переважно повторно використовуваною в *Java,* а *C* - відповідно в *C*. Незважаючи на те, що можливі взаємодія і запуск цими мовами одна одну, для *Java, C* і інших мов подібна поведінка не є типовою. У той же час будь-який додаток, який здатний підключитися до бази даних, може викликати *PL/SQL*. *Web*-служби і *CORBA* дозволяють виконувати одне і те ж для *Java* і *C,* але ці технології вводять свій власний додатковий рівень складності. Безумовно, відсутні причини, за якими *web*- служби не варто було б писати на *PL/SQL*! Якщо служба виконує якусь обробку даних, вона буде набагато ефективніше і надійніше працювати, якщо написати її на *PL/SQL*.

Якщо є можливість підключення до бази даних, можна застосовувати існуючий *PL/SQL*. Якщо для конфігурації, налаштування, використання бази даних застосовується в основному *API*, то цей *API* повинен бути заснований на *PL/SQL*. *Oracle* використовує *PL/SQL* в роботі з реплікацією, потоками, розкладами завдань, управлінням планування ресурсів, функціональністю *OLAP* і т.д. *PL/SQL* є універсальною мовою, за допомогою якої можна спілкуватися з базою даних.

Як і при роботі з іншими мовами, можна написати хороший код, а можна написати і погано працюючий код. *PL/SQL* не є чарівною мовою. При її

використанні не складно допустити точно такі ж помилки, як і при написанні коду на *Java, C, Visual Basic* або інших мовах. Можна створити циклічність (але база даних дозволяє встановити профілі ресурсів, які будуть обривати запропонований код, не дозволяючи йому працювати в базі даних). Можна написати код з логічними помилками. Однак багато помилок, які не важко допустити при роботі з іншими мовами, складно або ж неможливо здійснити при роботі з *PL/SQL*. Нижче наведені деякі приклади:

- **Застосування змінних прив'язки.** В *PL/SQL* неможливо не використовувати змінні прив'язки при роботі зі статичним *SQL*. Змінні прив'язки можна не використовувати тільки при виконанні динамічного *SQL* в *PL/SQL*, але й навіть тоді в *PL/SQL* набагато легше писати код із застосуванням змінних прив'язки, аніж без них. В інших мовах часто буває простіше не використовувати змінні прив'язки. А, як говорилося в попередніх розділах, застосування змінних прив'язки є ключем до масштабованості бази даних і до високої продуктивності її роботи.
- **Один раз розібрати, багато раз виконати.** При використанні статичного *SQL* неможливо провести повторний розбір оператора в *PL/SQL*. Для того, щоб це було можливо, необхідно програмувати із застосуванням динамічного *SQL*. *PL/SQL* автоматично прозоро поміщає оператор в кеш, забезпечуючи його одноразовий розбір і багаторазове виконання. При використанні динамічного *SQL* можна проводити розбір при кожному виконанні оператора, але буде коректніше навіть за допомогою динамічного *SQL* проводити розбір оператора один раз, а виконувати його багаторазово в *PL/SQL*.
- ***SELECT **** В *PL/SQL* є можливість безпечного використання *SELECT **. Такий підхід не є кращим, оскільки надає негативний вплив на продуктивність (завжди краще вибрати тільки один необхідний стовпець). Але у всіх інших мовах не можна помістити *SELECT* * в будь-яку програму! Повторне створення таблиці з певними в іншому порядку стовпцями, додавання або вилучення стовпця може привести до ушкодження додатка (але не в *PL/SQL*). Якщо код написаний вірно, то *PL/SQL* самостійно внесе зміни.

- **Зміни в схемі бази даних.** Припустимо, що додаток змінює тип даних стовпця *COMMENTS* з *VARCHAR2(80)* на *VARCHAR2(255).* Для того щоб працювати з внесеними змінами, деякі існуючі додатки, написані не на *PL/SQL*, повинні бути перепрограмовані. Але написаних належним чином на *PL/SQL* додатків це не торкнеться.
- **Я не знав, що ви це використовуєте.** Ця помилка трапляється через ненадійність відстеження управління. Якщо комусь невідомо, що конкретний об'єкт використовується вами, то він вважає можливим його змінити. При застосуванні *PL/SQL* інформація про те, хто, що і де використовує, зберігається в словнику даних! Можливість того, що «Я не знав», дуже сильно зменшуються.

PL/SQL є не тільки найбільш продуктивним середовищем для створення орієнтованих на дані програм, що викликаються з різних мов, але і захищає ці програми. Якщо використовується *PL/SQL*, значно складніше допустити деякі поширені помилки програмування в *Oracle*.

Далі ми розглянемо декілька важливих моментів, що стосуються програмування/розробки *PL/SQL*. Серед них є питання, пов'язані з основними принципами програмування, наприклад: «слід писати якомога менше коду». Інші теми торкнуться безпосередньо мови *PL/SQL*: використання *%TYPE* і *%ROWTYPE*.

3.2.4. Слід писати якомога менше коду

Основним завданням є написання якомога меншого обсягу коду. У реляційній базі даних процедурний код слід використовувати лише тільки після того, як буде доведено, що не працює метод, заснований на наборах. Нерідко зустрічається подібний процедурний код:

```
Begin
    For x in (select * from table@remote_db)
    Loop
        Insert into table (c1, c2, ...) values (x.c1, x.c2, ...);
```

End loop;

End;

Але не існує причин, з яких цей код не можна було б написати так:

- *Insert into table (c1, c2, ...) select c1, c2, ... from table@remote_db*

Основне завдання полягає в тому, щоб зробити це в *SQL*, а PL/*SQL* використовувати тільки тоді, коли *SQL* не може цього виконати.

Часто зустрічається і така методика написання коду:

*For a in (select * from tl)*

Loop

*For b in (select * from t2 where t2.key = t1.key)*

Loop

*For c in (select * from t3 where t3.key = t2 .key)*

Loop

…

В даному випадку розробник не захотів «обременять» базу даних з'єднанням, не усвідомлюючи, що подібний підхід працює у багато разів повільніше, ніж такий:

*For x in (select **

From tl, t2, t3

Where tl.key = t2.key

And t2.key = t3.key) loop

3.2.5. Використання пакетів

Замість одиничних процедур або функцій слід використовувати пакети. Пакети є єдиним способом кодування в PL / SQL і застосовуються в проектах будь-яких масштабів.

У пакетів є наступні переваги в порівнянні з автономними процедурами і функціями:.

- *Вони розривають ланцюжок залежних з'єднань*. Пакети скорочують або видаляють наслідки каскадної недійсності.

- *Вони збільшують простір «присваеваемых имен».* Зазвичай тільки одна процедура може мати ім'я *P.* Тепер же таких процедур може бути десяток. В єдиному пакеті може міститися багато процедур. Буде існувати один об'єкт словника даних - пакет - замість окремого об'єкта словника для кожної даних процедури / функції.

- *Вони підтримують перезагрузку.* В єдиному пакеті може міститися кілька однаково названих процедур, кожна з яких приймає інший набір вхідних даних.

- *Вони підтримують інкапсуляцію***.** Я написав багато маленьких підпрограм, які не використовуються за межами пакета. Я сховав їх у пакет так, що їх не видно за межами пакета, і тільки один я можу їх бачити.

- *Вони підтримують постійні змінні в сесії.* Можна мати змінні, які зберігають свої значення в базі даних від виклику до виклику.

- *Вони підтримують удосконалення або настройку коду під час запуску.* Пакет може містити частини коду, які виконуються з першим посиланням на пакет в сесії. Тобто пакет дозволяє мати складний автоматично виконуємий код ініціалізації.

- *Вони дозволяють групувати разом пов'язані функції.* В результаті групування програм частини коду використовуються спільно.

Найбільш важливою причиною використання пакетів є те, що з їх допомогою розривається ланцюжок залежних з'єднань в базі даних.

Висновки до розділу

Весь код повинен бути розміщен на екрані. При написанні коду слід дотримуватися наступного правила: програма (метод, процедура, функція або щось ще, що требуслея вся кликати) повинна уміщатися на екрані. Логіка повинна легко відслідковуватися від початку до кінця на моніторі. Це правило навчає думати програмістів модулно, оскільки доведеться розділити код на невеликі частини, легчі для розуміння. Будь-який процедурний код без *SQL* завдовжки більше 80 рядків слід розбивати!

Для якісної роботи коду слід застосовувати пакети. Вони забезпечують необхідну програмну структуру, таку як інкапсуляція і зскорочення простору привласнюваних імен, а також сприяють застосуванню модульного кодування. Дуже важливо, що пакети розривають ланцюжок залежних з'єднань, працюючи так, що зміни в схемі бази даних не призводять до того, що вся схема або навіть база даних становиться недійсною і вимагає перекомпіляції. У яких же випадках прийнятно використання одиночних процедур і функцій? Вони підходять для одиночних утиліт, які ніколи не визиваються іншим кодом. Ось деякі з цих інструментів:

- Сценарій *SHOW_SPACE*, який показує що використовуєме об'єктом простір.
- Утиліта *PRINT_TABLE*, яка посторінково виводить в *SQL * Plus* результуючі набори.
- Процедура *DUMP_CSV (PL / SQL)*, яка бере будь-який запит і створює файл зі значеннями, відокремленими комами.

Поодинокі функції і процедури також прийнятні для швидкої демонстрації функціональності і для тестування. Але для всіх робочих кодів в системі пакети є єдиним рішенням.

ВИСНОВКИ

У висновку хочу зазначити що дипломна робота напряму впливає на актуальну на сьогоднішній день тему на ринку, а саме оптимізація додатку, що використовує архітектуру клієнт-сервер, через оптимізацію бази даних на платформі *Oracle*.

Було проведено аналіз архітектурних рішень, що впривають на продуктивність бази даних. Було досліджено налашування, які використовуються при проектуванні ефективної схеми, проведено тестування оперативної пам’яті під час критичних навантажень та було проведено теоретичне дослідження на масштабованість.

Найбільш суттєвими науковими результатами дипломного проекту можна вважати:

– Порівняння існуючих методів проектування ефективних схем *Oracle*.

– Обгрунтування переваг та недоліків даних методів, які включають як вибір апаратної частини так і програмних налаштувань.

– Сформування рекомендацій щодо ефективного використання цих методів при проектуванні схем.

– Сформування рекомендацій щодо будування високоефективних запитів на мові програмування *PL/SQL*.

Розуміючи що тема вибору правильних архітектурних рішень при проектуванні ефективної схеми є дуже важливою для будь-якої організації, яка вирішить зберігати важливу для неї інформацію в реляційних базах даних, в роботі також було досліджено питання економічного характеру, які актуальні сьогодні на ринку і в подальшому впливають на продуктивність та масштабованіть бази даних.

У подальшому завдяки даним дослідженням та рекомендаціям можно розширювати або перебудовувати додатки, з архітектурою клієнт-сервер, які використовують реляційні бази даних, як основне сховище зберігання важливої інформації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Белоногов Г.Г. Автоматизация процессов накопления, поиска и обобщения информации / Г.Г. Белоногов, А.П. Новоселов – М.: Наука, 1979. – 256 с.

2. *Bloch J. Effective Java Library of Congress,* 2012 – 413 с.

3. Сергеев. С.Ф. Методы тестирования и оптимизации интерфейсов инфомационных систем. 2013. – 115 с.

4. Глушаков С.В. Практикум по *C*++/ Глушаков С.В., Смирнов С.В., Коваль А.В. - Х.: ФОЛИО, 2006. - 525 с.

5. Денисов А.А. Теория больших систем управления / А.А. Денисов, Д.Н. Колесников – Л.: Энергоиздат, Ленинград. отделение, 1982. – 288 с.

6. Дейл Н. Программирование на С++/ Дейл Н., Уимз Ч., Хедингтон М. - Пер. с англ. - М.: ДМК, 2000.- 672 с.

7. Дейт К. Введение в системи баз данных / Дейт К. - М., 2005. – 980 с.

8. Барсенгян, А.А., Анализ данных и процессов [Текст] / А. А. Барсенгян. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. – 512 с.

9. Ашманов, И. С. Оптимизация и продвижение сайтов в поисковых системах [Текст] / И. С. Ашманов. — СПб: Питер Пресс, 2014. — 463 с.

10. Евстифеев А.В. Микроконтролеры *AVR* семейства *Classic* фирмы "*ATMEL*" / Евстифеев А.В. - М.: Издательский дом "Додэка-*XXI*", 2002.- 288 с.

11. Жуков І.А. Комп'ютерні мережі та технології: Навч. посібник/ Жуков І.А., Гуменюк В.О., Альтман І.Є. - К.: НАУ, 2004.- 276 с.

12. Чекалов А. Прагматический подход к разработке приложений *Web* баз данных / Чекалов А. – М.: Питер, 2008. – 246 с.

13. Ландэ, Д. В. Интернетика: навигация в сложных сетях: модели и алгоритмы [Текст] / Д. В. Ландэ, А. А. Снарский., И. В. Безсуднов. — М.: Либроком (*Editorial URSS*), 2009. — 264 с.

14. Гасанов, Э. Э., Кудрявцев, В. Б. Теория хранения и поиска информации [Текст] / Э. Э. Гасанов, В. Б. Кудрявцев. — М.: Физматлит, 2002. — 288 с.

15. Юринець В. Комп'ютерні мережі. Інтернет: Навч. Посібник / В.Юринець, Р.Юринець. - Львів: ЛНУ ім. І.Франка, 2006. - 524 с.

16. Автоматизовані системи управління і нові інформаційні технології: Збірник наукових праць. Вип.2. - К.: Академперіодика, 2004. - 152 с.

17. Хомоненко А.Д. Базы данных: Учебник для высших учебных заведений – Издание второе, дополненное и переработанное / Хомоненко А.Д., Цыганкова В.М., Мальцев М.Г. – СПб.: КОРОНА принт, 2011. – 668 с.

18. Шафрин Ю.А. Основы компьютерной технологии / Шафрин Ю.А. - М., 2008. – 246 с.

19. Кулик М.С. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. / М.С. Кулик, А.В. Полухін – К.: НАУ, 2011. – 42 с.

20. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения: ГОСТ 19.701-90. – Введ. 01.01.92. – М.: Изд-во стандартов, 1991. – 25 с