Факультет комп'ютерних наук та кібернетики
Київського національного університету
імені Тараса Шевченка

Кулябко П.П.

Проблеми рефакторінгу баз даних
Частина 2.

Навчально-методичний посібник

Київ
2023

УДК 681.3

Кулябко П.П. Проблеми рефакторінгу баз даних. Частина 2:

Навчальний посібник для студентів факультету комп’ютерних наук та

кібернетики. –

Київ. – 2023. – 77 с.

Затверджено вченою радою

факультету комп'ютерних наук

та кібернетики,

протокол № 11 від 30 травня 2023 р.

Перевірка працездатності

Перед серйозною роботою по рефакторінгу слід перевірити кілька пунктів, особливо якщо існує значна кількість великих, ресурсоємних запитів. Іноді грубі помилки, які легко виправити, залишились непоміченими через обмеження часу, незнання або просте нерозуміння роботи СУБД. У цьому розділі буде розглянуто декілька питань, які слід контролювати, і, якщо необхідно, виправити, перш ніж приймати висновки щодо рефакторінгу. При певній удачі рефакторінг може видатися не таким вже й жахливим процесом.

Одним з найважливіших аспектів продуктивності СУБД щодо окремих запитів є *ефективність оптимізатора запитів* (на жаль, оптимізатор мало що може зробити для поганих алгоритмів). Оптимізатор базує свій пошук кращого плану виконання на ряді факторів: як правило, на тому, що словник даних говорить про існуючі індекси, а також про статистику, що зберігається в словнику даних, збір якої зазвичай є рутинною частиною адміністрування бази даних.

Існує дві речі, які потрібно перевірити перш за все, коли виникають проблеми з ефективністю для певних запитів: чи достатньо актуальна та деталізована статистика, і чи є індексація доречною. Це дві передумови для оптимізатора виконувати свою роботу належним чином (незалежно від того, чи дійсно оптимізатор виконує свою роботу належним чином, коли індексація та статистика не такі, як треба, - це інше питання). Перш за все розглянемо тему статистики, включаючи статистику індексів, перш ніж коротко обговорити індексацію. Цей порядок може дивувати, тому що багато людей прирівнюють хорошу продуктивність SQL до правильного використанням індексів. Насправді, хороша продуктивність має багато спільного з використанням підходящого індексу, як і з невикористанням поганого індексу.

Статистика та проблеми з даними

Статистика - це загальний термін, який охоплює велику кількість даних, які СУБД накопичує стосовно того, що вона зберігає. Простір, необхідний для зберігання статистики, незначний, але час, необхідний для їх збирання, навпаки, значний; збір статистики може бути або автоматизований, або викликаний адміністраторами. Оскільки збір детальної статистики може бути дуже важкою операцією (в гіршому випадку, ви можете сканувати всі дані у вашій базі даних), ви можете попросити більш-менш уточнені статистичні дані, обчислені або оцінені; різні продукти СУБД збирають дещо іншу інформацію.

Доступна статистика

Було б досить нудно детально розглядати всі статистичні дані, які збирають різні продукти та які можуть бути великими у випадку Oracle і SQL Server. Давайте просто розглянемо, що може бути корисною інформацією для оптимізатора при розгляді альтернативних планів виконання.

По-перше, слід розуміти, що під час сканування таблиці або доступу до неї за допомогою індексу існують відмінності. Те, як SQL-движок читає рядки, також відрізняється, і є багато роботи з фізичним зберіганням даних. Коли СУБД зберігає дані, вона виділяє пам’ять для таблиці, а розподіл пам'яті не є чимось неперервним, а дискретним: великий шматок резервується у файлі для конкретної таблиці, а коли цей шматок заповнений, то резервується інший. Ви не виділяєте пам’ять з кожним вставленим рядком. Як наслідок, дані, які належать до таблиці, фізично кластеризовані, тими розмірами, які залежать від вашого

носія даних (якщо ви використовуєте дискові масиви, таблиця не буде кластеризована, оскільки може бути в одному розділі одного диска). Але коли СКБД сканує таблицю, вона зчитує багато даних за один раз, тому що певний фрагмент пам'яті, де зберігається один рядок, має дуже високий шанс зберігати рядки, які будуть потрібні у подальшому.

На відміну від цього, індекс асоціюється з тими ж рядками ключів індексу, які не обов'язково близькі один до одного (якщо це не кластерний індекс SQL Server, чи не таблиця, організована за допомогою індексів Oracle, або подібний тип сховища, який СУБД знає як обробляти). Коли СУБД шукає дані, то адресна одиниця - це *page* (яка називається *block* Oracle) з кількох кілобайтів. Один пункт індексу, пов'язаний із значенням ключа, повідомить нам, що є рядок, який відповідає ключу на одній сторінці, але, ймовірно, жоден інший рядок на сторінці не відповідатиме тому ж ключу. Пошук буде працювати на основі рядок-за-рядком набагато довше, ніж у випадку повних сканувань. Група індексів буде відмінною для відносно невеликого числа рядків, але не обов'язково так добре, як велика група повних сканів з великою кількістю рядків.

В результаті, при фільтрації рядків і застосуванні підфрази where, головна увага оптимізатора у вирішенні того, чи може індекс бути корисним - це селективність, тобто, який відсоток рядків задовольняє даному критерію. Це вимагає знання двох величин: кількості рядків у таблиці та кількості рядків, які проходять критерій пошуку.

Наприклад, наступну умову можна більш-менш легко оцінити:

where my_column = *some_value*

Коли my_column індексується з властивістю unique, нам не потрібна статистика: умова буде задоволена або одним рядком, або без рядків. Стає трохи цікавіше, коли стовпець індексується з властивістю nonunique. Щоб оцінити селективність, ми повинні покладатися на таку інформацію, як кількість різних ключів в індексі; так ви можете, наприклад, обчислити, що якщо ви є Han Chinese, то ваше прізвище становить одне з приблизно трьох тисяч, а оскільки Han Chinese складають 92% населення Китаю, одна прізвище розподіляється в середньому на 400 000 чоловік. Якщо ви думаєте, що 1: 3,000 - це селективне співвідношення, то ви можете протиставити китайські імена французьким іменам, з яких близько 1,3 мільйона (включаючи імена іноземного походження) для населення близько шістдесяти мільйонів, або 46 осіб на ім'я в середньому.

Селективність є серйозною темою, а переваги доступу до індексів над звичайним скануванням таблиць іноді є сумнівними при низькій селективності. Щоб проілюструвати цей момент, була створена таблиця з п'яти мільйонів рядків з п'ятьма стовпчиками - C0, C1, C2, C3 і C4 - і заповнені стовпці відповідно випадковим значенням від 1 до 5, випадкове значення від 1 до 10 , випадкове значення від 1 до 100, випадкове значення від 1 до 1000 і випадкове значення від 1 до 5000. Потім запускалися запити на одне конкретне значення для кожного стовпця, запити, які повертають від 1 000 000 до 1000 рядків залежно від стовпця, який використовується як критерій пошуку. Ці запити запускалися кілька разів, спочатку на неіндексованій таблиці, а потім після індексування кожного стовпчика. *(Only regular indexes, not clustered indexes) Крім того, статистика не обчислювалася.

Таблиця 1 підсумовує поліпшення продуктивності, яке було отримано з використанням індексів. Значення - це відношення часу, необхідного для запуску запиту по неіндексовавному стовпцю до часу виконання запиту по індексованому стовпцю; значення менше 1 вказує на те, що потрібне більше часу для вибірки рядків, коли існував індекс, аніж тоді, коли його не було.

Таблиця 1 . Зростання продуктивності завдяки індексуванню

Результуючі рядки	**MySQL**	**Oracle**	**SQL Server**
~ 1,000,000	x0.6	x1.1	x1.7
~ 500,000	x0.8	x1.0	x1.5
~ 50,000	x4.3	x1.0	x2.1
~ 5,000	x39.2	x1.1	x20.1
~ 1,000	x222.3	x20.8	x257.2

Єдиним продуктом, для якого у цьому прикладі пошуки по індексу є набагато кращими ніж повні сканування, є SQL Server. У випадку MySQL у цьому прикладі відчути позитивний ефект можна тільки тоді, коли повертається 1% рядків; коли повертається 10% або більше рядків, ефект індексу є дуже шкідливим. У випадку Oracle, чи є індекс чи ні, це досить несуттєво, поки ми не повернемо 0,02% рядків. Навіть з SQL Server ми не отримуємо поліпшення порядку величини (у 10 разів), поки кількість рядків не зменшиться до дуже скромного розміру таблиці.

Немає жодного магічного відсотка, який би підходив для комфортного жорсткого правила, наприклад, "з Oracle, якщо ви повертаєте менше, ніж магічний відсоток рядків, ви повинні мати індекс". Можуть впливати багато факторів таких, як впорядкування рядків у таблиці (індекс на стовпці зі значеннями, які збільшуються в ході вставки рядків, є, наприклад, більш ефективним для запиту, ніж індекс подібної селективності на стовпчику, значення якого випадкові). Це не тому, що стовпець, який використовується в якості критерію пошуку, повинен бути індексованим (а іноді й не повинен).

Селективність важлива, але, на жаль, селективність - це ще не все. Заповнення п’ятимільйоннорядкової таблиці було випадковими генерованими значеннями, а випадкові генератори виробляють рівномірно розподілені числа. У реальному житті розповсюдження найчастіше рівномірне. Попереднє порівняння між прізвищами в Китаї та у Франції, можливо, спровокувало таке питання: навіть у Франції в середньому приховуються вражаючі відмінності. Якщо ви француз, а ваше ім'я - Martin, ви поділяєте своє ім'я з більш ніж 200 тисячами інших французів (не кажучи вже про англосаксів); навіть якщо ваше ім'я Mercier, то більше 50 тисяч французів мають те ж саме ім’я. Звичайно, це все ще досить оброблювані числа порівняно з іменами Li, Wang і Zhang, які разом складають близько 250 мільйонів людей в Китаї.

Той факт, що ви не можете припустити рівномірний розподіл значень даних, означає, що під час пошуку певних значень індекс безпосередньо перейде до потрібного набору результатів, а коли ви шукаєте інші значення, це не так ефективно. Рейтинг ефективності індексу по відношенню до певної величини особливо важливий, коли у вас є кілька критеріїв і кілька корисних індексів, з яких ніхто не виступає явним переможцем з

точки зору середньої селективності. Щоб допомогти оптимізатору зробити усвідомлений вибір, вам може знадобитися збирати гістограми. Для обчислення гістограми СУБД ділить діапазон значень для кожного стовпця на декілька категорій (одна категорія на значення, якщо у вас менше двох сотень різних значень, в іншому випадку - фіксована кількість) і підраховує, скільки значень потрапляє до кожної категорії. Таким чином, оптимізатор знає, що під час пошуку людей, які мають певне ім'я та які народилися між двома датами, він скоріше використовує індекс імені, оскільки це рідкісне ім'я, а коли ви шукаєте інше ім'я для цього ж діапазону дат буде використовувати індекс на дату, тому що ім'я дуже поширене. Це основна ідея гістограм, хоча пізніше в цьому розділі ви побачите, що гістограми можуть опосередковано приносити власні проблеми з продуктивністю.

Крім кількості рядків у таблиці, середньої селективності індексів і (при необхідності) способу розподілу значень можуть бути дуже корисними інші відомості. Вже згадувалось про зв’язок між порядком ключів індексу і порядком рядків таблиці (який не може бути більш сильним, якщо у вас є кластерний індекс з SQL Server або MySQL); якщо цей зв'язок є сильним, індекс буде ефективним при пошуку в діапазоні ключів, оскільки порівнювані рядки будуть кластеризовані. Багато інших значень статистичного характеру можуть бути корисні оптимізатору для оцінки альтернативних шляхів виконання і для оцінки кількості рядків (або *cardinality*), які залишаються після кожного наступного шару умов відбору. Ось список деяких із цих значень:

Кількість null значень

Якщо null значення не зберігаються в індексі (як у випадку з Oracle), знання того, що більшість значень у стовпці є null *(може бути результатом неправильного проектування таблиці), інформує оптимізатор, коли стовпець згадується при прийнятті заданого значення, то умова, ймовірно, дуже селективна, навіть якщо в індексі є декілька різних ключових значень.

Область значень

Знання того, в якому діапазоні знаходяться значення в стовпці, дуже корисно при спробі оцінити, скільки рядків може бути повернене за умовою, наприклад >=, <=, або between, зокрема, коли немає гістограми. Крім того, умова рівності величині, що виходить за межі відомого діапазону, ймовірно, буде дуже селективним. Якщо стовпці індексуються, то мінімальне та максимальне значення можна легко отримати з індексу.

Перевірка наявності статистики проста. Наприклад, за допомогою Oracle ви можете запустити під

SQL*Plus такий сценарій:

```
col "TABLE" format A18
col "INDEX" like "TABLE"
col "COLUMN" like "TABLE"
col "BUCKETS" format 999990
break on "TABLE" on "ANALYZED" on sample_size on "ROWS" on "INDEX" on
"KEYS"
select t.table_name "TABLE",
        to_char(t.last_analyzed, 'DD-MON') "ANALYZED",
        t.sample_size,
        t.num_rows    "ROWS",
        i.index_name "INDEX",
        i.distinct_keys "KEYS",
```

```
        i.column_name           "COLUMN",
        i.num_distinct "VALUES",
        i.num_nulls     "NULLS",
        i.num_buckets "BUCKETS"
from user_tables t
    left outer join (select i.table_name,
                            i.index_name,
                            i.distinct_keys,
                            substr(ic.column_name, 1, 30) column_name,
                            ic.column_position pos,
                            c.num_distinct,
                                c.num_nulls,
                               c.num_buckets
                        from user_indexes i
                                inner join user_ind_columns ic
                                        on ic.table_name = i.table_name
                                       and ic.index_name = i.index_name
                                inner join user_tab_columns c
                                        on c.table_name = ic.table_name
                                       and c.column_name = ic.column_name) i
                        on i.table_name = t.table_name
        order by t.table_name,
                i.index_name,
                i.pos
        /
```

Або з MySQL (де є набагато менше даних для перевірки), ви можете запустити наступне

```
        select t.table_name,
                t.table_rows,
                s.index_name,
                s.column_name,
                s.cardinality
```

```
from information_schema.tables t
    left outer join information_schema.statistics s
                    on s.table_schema = t.table_schema
                   and s.table_name = t.table_name
where t.table_schema = schema( )
order by t.table_name,
         s.index_name,
         s.seq_in_index;
```

Процес є набагато складнішим для SQL Server, для якого потрібно спочатку викликати sp_helpstats з назвою таблиці та 'ALL', а потім запустити dbcc show statistics для кожної статистики, назва якої повертається попередньою збереженою процедурою. Але що може бути цікавим - це перевірка чи є аномалії, які можуть спонукати оптимізатор запитів до помилки.

Пастки для Оптимізатора

Нерівномірний розподіл значень у стовпці є вагомою причиною для оптимізатора обчислювати витрати виконання з неправильних припущень, а отже, встановити неправильний план виконання. Але можуть бути й інші, менш очевидні випадки, коли оптимізатор збивається.

Екстремальні значення

Може бути корисно перевірити діапазон значень для індексованого стовпця, це легко зробити так:

```
select min(column_value), max(column_value)
from my_table;
```

Коли є індекс, мінімальне та максимальне значення можуть бути прочитані дуже швидко з індексу. Такий запит може привести до деяких цікавих відкриттів. В Oracle, наприклад, досить часто знаходяться значення в стовпцях дати, що стосуються першого століття нашої ери, тому що люди помилково ввели дати як MM/DD/YY, коли Oracle очікував MM/DD/YYYY (ця проблема набагато рідше стається з SQL Server, для якого тип дати починається з 1753 або з MySQL, який приймає дати з 1 січня 1000 і далі). З числовими стовпцями ми можемо мати такі значення, як –99999999, які використовуються для обходу того, що стовпець є обов'язковим, а значення невідоме (питання дизайну, очевидно) і так далі. Майте на увазі, що, коли немає іншої інформації, оптимізатор припускає, що значення рівномірно розподілені між найменшими і найбільшими значеннями, і можуть сильно переоцінювати або недооцінювати кількість рядків, що повертаються скануванням діапазону (умова нерівності). В результаті він може прийняти рішення використовувати індекс (якщо це не треба) або може зробити

навпаки. Виправлення несуттєвих (або поганих) значень - це перший крок, який може знову повернути оптимізатор на коректний шлях.

Те, що стосується найменших значень, також справедливо для найбільших значень. У колонці дат звичайно використовуються «далекі, далекі дати» для того, щоб означати «невживані» або «поточні». Ви також можете перевірити наявність аномалій, що екстремальні значення нічого не скажуть вам. Наприклад, два програмісти можуть використовувати різні стандарти для «далеких, далеких дат»: один програміст може використовувати 31 грудня 2999, тоді як інший програміст використовує рік 9999. Якщо ви коли-небудь зіткнетеся з таким випадком, то ваш оптимізатор переживе великі зусилля, щоб зрозуміти перевантажений діапазон дат. Навіть при гістограмах два різні піки на віддалених датах можуть дати оптимізаторові неправильні уявлення про розподіл. Всякий раз, коли оптимізатор стикається із значенням, на яке немає явного посилання в гістограмі, він повинен екстраполювати його розподіл через неперервну функцію, а дуже нерегулярний розподіл може призвести до невірної екстраполяції. І, зрозуміло, помилки тільки і чекають, коли один розробник буде перевіряти одне значення, а інший розробник встановив інше. Ви можете легко протестувати такий випадок, запустивши щось на зразок наступного (на жаль, це важкий запит, який слід запускати на копії робочих даних):

```
select extract(year from date_column), count(*)
from my_table
group by extract(year from date_column)
```

Навіть, якщо у вашій базі даних використовується лише одна дата далекого майбутнього, то використання такої віддаленої

дати для задання поточного значення потребує гістограм. В іншому випадку, як тільки дані накопичуватимуться і минулі історичні значення будуть представляти значну вагу в базі даних, тоді кожен перегляд діапазону минулих значень - тобто, запит, в якому ви маєте щось таке:

```
where expired_date < now( )
```

при відсутності гістограми БД буде переглядати, ніби вона повертала, скажімо, десятиліття з одного або двох тисячоліть. Якщо ви дійсно мали дані протягом одного тисячоліття, то десятиліття буде становити близько 1% всіх рядків, якщо ви дійсно хочете повертати 99% всіх рядків вашої таблиці. Оптимізатор, можливо, захоче використовувати індекс тоді, коли повне сканування буде більш ефективним, або він може вибрати індекс стовпця дати замість іншого більш ефективного індексу.

Тимчасові таблиці

Варто також підкреслити, що тимчасові таблиці, зміст яких, за визначенням, дуже мінливий, також багато в чому сприяють пікантності життя оптимізатора. Це особливо вірно, коли єдина річ, яка вказує на їх тимчасовий характер - використання TMP як префікса або суфікса в їх імені, і якщо вони не були створені як тимчасові таблиці, але використовуються (важливий нюанс) тільки як тимчасові таблиці. На жаль, корисні натяки в назві повністю втрачені на оптимізаторі. Як наслідок, будь-яка таблиця, створена як звичайна таблиця, розглядається як звичайна таблиця, незалежно від її назви і до неї застосовуються основні правила. Якщо, наприклад, статистику було обчислено, коли таблиця була порожньою, оптимізатор буде дотримуватися ідеї, що таблиця містить нуль або дуже мало рядків, незалежно від того, що відбувається, якщо ви не попросите СУБД зберігати

таблицю під постійним спостереженням, так Ви можете робити в Oracle, коли задаєте динамічне квантування або в SQL Server з автоматичною статистикою, що явно викликає накладні витрати.

У деяких продуктах тимчасові таблиці видно лише підчас сесії, в ході якої вона була створена; отже, просто наявність таблиці, яка містить TMP або TEMP у своїй назві в information_schema.tables, заслуговує на дослідження (насправді, використання тимчасових таблиць саме по собі заслуговує на дослідження; воно може бути виправданим, але іноді це маскує недоліки SQL).

Огляд індексування

Перевірка того, чи правильно проіндексовані таблиці, ймовірно, є одним з перших кроків, які потрібно виконати. Проте результати в табл.1 переконують нас у тому, щоб не слідувати простому правилу «це повільно, тому ми повинні додати індекси». Власне, (і досить дивно) індекси частіше зайві, ніж недостатні.

Перший заклик до поліпшення продуктивності програми, як правило, є питанням наведення порядку в індексації. Індекси часто недостатньо вивчені, і в цьому розділі я наведу кілька прикладів, які допоможуть краще зрозуміти, як вони працюють, і як можна визначити, чи потрібно переглядати індексацію.

По-перше, існують два типи індексів: індекси, які існують як наслідок розробки бази даних, та індекси продуктивності.

Індекси, які випливають з проекту бази даних, є індексами, що забезпечують застосування семантики - індекси на первинних ключах і стовпці з унікальними значеннями. Коли ви визначаєте обмеження на таблицю, яка говорить, що деякий набір стовпців

однозначно ідентифікує рядок у стовпці, ви повідомляєте СУБД, що вона повинна перевірити, для кожної вставки, що рядок не існує і що він повинен повернути помилку, якщо ключ дублює вже існуючий ключ. Перевірка наявності ключа в дереві під час вставки є дуже ефективною, і трапляється, що індекси, найчастіше, є деревоподібними структурами. Ці індекси є, таким чином, особливостями реалізації. Оскільки стовпці, які однозначно ідентифікують рядок, часто використовуються для пошуку та повернення цього рядка, особливість реалізації подвоюється як досить зручний показник продуктивності, і все це на краще. Деякі продукти (такі як движок InnoDB з MySQL) також вимагають індекс на foreign ключах, що не вимагає ні Oracle, ні SQL Server. Хоча індексація foreign ключів зазвичай рекомендується як "хороша практика", це не завжди так. По-перше, у багатьох випадках foreign ключ - це перший стовпець первинного ключа, який вже індексується індексом первинного ключа. Але справжнє питання полягає в тому, яка таблиця повинна керувати запитом - таблиця, що містить foreign ключ чи таблиця посилання, оскільки вимоги до індексації різні. Більш детально запити будуть обговорюватись в подальшому. Поки що скажемо, що будь-який індекс на foreign ключі, коли він не є обов'язковим, слід розглядати як індекс продуктивності.

Індекси продуктивності - це індекси, які створюються для швидшого пошуку; вони включають в себе регулярні індекси B-дерева, та інші типи індексів, такі як bitmap індекси і hash індекси, їх можна зустріти в багатьох місцях. (Сюди не включаються повнотекстові індекси в цьому обговоренні, що може бути дуже цікавим з точки зору продуктивності, але дещо поза ядром СУБД.) Заради простоти, далі будемо говорити про звичайні індекси.

Ми бачили, що регулярні індекси іноді роблять пошук швидшим, іноді роблять пошук повільнішим, а іноді не мають ніякого значення. Оскільки вони підтримуються в реальному часі, і оскільки кожна вставка рядків, видалення рядків або оновлення індексованого стовпця означає вставку, видалення або оновлення індексованих ключів, тому будь-який індекс, який не має значення для пошуку, є фактично чистою втратою продуктивності. Втрата є більш серйозною, коли виникає конфлікт при вставці, тому що не так просто поширювати паралельні вставки на деревоподібний індекс, як над таблицею; у деяких випадках запис до індексів є справжнім вузьким місцем. Деякі індекси необхідні, але індексація кожного стовпця про всяк випадок не є, м'яко кажучи, дуже розумною ідеєю. На практиці погана індексація є набагато більш поширеною, ніж вона повинна бути.

Швидкий огляд схеми індексації

Перевірка того, як таблиці проіндексовані у схемі займає кілька секунд і може дати вам підказки про існуючі або очікувані проблеми з продуктивністю. Якщо у вашій схемі немає тисяч таблиць, виконання запиту на зразок наступного може бути дуже корисним:

```
select t.table_name,
    t.table_rows,
            count(distinct s.index_name) indexes,
            case
              when min(s.unicity) is null then 'N'
              when min(s.unicity) = 0 then 'Y'
              else 'N'
   end unique_index,
  sum(case s.columns
```

```
when 1 then 1
            else 0
            end) single_column,
        sum(case
              when s.columns is null then 0
              when s.columns = 1 then 0
              else 1
       end) multi_column
       from information_schema.tables t
            left outer join (select table_schema, table_name,
                                index_name, max(seq_in_index) columns,
                                min(non_unique) unicity
                                     frominformation_schema.statistics
                               where table_schema = schema( )
                               group by table_schema,table_name,index_name) s
                               on s.table_schema = t.table_schema
                               and s.table_name = t.table_name
       where t.table_schema = schema( )
       group by t.table_name, t.table_rows
       order by 3, 1;
```

Якщо ви використовуєте Oracle, то можете віддати перевагу наступному:

```
    select t.table_name,t.num_rows table_rows,
            count(distinct s.index_name) indexes,
            case
              when min(s.unicity) is null then 'N'
              when min(s.unicity) = 'U' then 'Y'
              else 'N'
                  end
unique_index,sum(case
          s.columns
               when 1 then 1
```

```
                                    else 0
                                  end) single_column,
                sum(case
                when s.columns is null then 0
                when s.columns = 1 then 0
                else 1
                  end) multi_column
    from user_tables t
        left outer join (select ic.table_name, ic.index_name,
                                max(ic.column_position) columns,
                                min(substr(i.uniqueness, 1, 1)) unicity
                           from user_ind_columns    ic, user_indexes i
                  where i.table_name = ic.table_name
                    and i.index_name = ic.index_name
                          group by ic.table_name,
                                   ic.index_name) s
                       on s.table_name =
    t.table_name
  group by t.table_name, t.num_rows
  order by 3, 1;
```

Цей запит відображає для кожної таблиці кількість рядків, кількість індексів, чи є unique індекс і скільки індексів є індексами з одним стовпчиком і скільки мультистовпчикових складних індексів. Такий запит не буде вказувати на кожну помилку, не дасть вам чудові уявлення про те, як магічно збільшити продуктивність, але може надати вам кілька підказок на потенційні проблеми:

Неіндексовані таблиці

Таблиці зі значним числом рядків і без індексу буде легко помітити. Такі таблиці можуть мати своє використання, зокрема

під час завантаження бази даних або масових операцій, що включають перетворення даних. Але вони також можуть приховувати великі помилки.

Таблиці без жодного unique індексу

Таблиця без будь-якого unique індексу, як правило, не має обмеження по первинному ключу. Відсутність первинного ключа - це погано для даних, оскільки СУБД не буде робити нічого, щоб запобігти вставці дублів рядків.

Таблиці з одним індексом

Відзначимо, що наявність лише одного unique індексу в таблиці не є, само собою, сертифікатом хорошого проекту. Якщо індекс первинного ключа є індексом з одним стовпчиком, (і якщо цей стовпець є автоматично заповнюваним зростаючими числами, які генеруються системою,), то це майже так само погано, як і відсутність індексу. СУБД не буде більш корисним для запобігання вставці дубльованих рядків, тому що всякий раз, коли ви знову вставляєте ті ж самі дані, вона згенерує новий номер, який зробить новий рядок іншим від інших. Використання системного числа може бути виправдано як зручний короткий ключ для використання в інших місцях для посилання на рядок, але в 99% випадків воно повинне бути доповнене другим unique індексом, який знаходиться на стовпцях, який дійсно дозволяє стверджувати, що певний рядок є особливим і ніби іншим.

Таблиці з багатьма індексами

Дуже велика кількість індексів рідко є хорошим знаком, за винятком контексту даних, де це є виправданою і стандартною практикою. У звичайній, робочій базі даних (на відміну від бази даних підтримки прийняття рішень), маючи майже стільки ж

індексів, скільки у вас стовпці, не потрібно. Цілком імовірно, що принаймні деякі з індексів у кращих випадках є марними, а в гірших - шкідливими. Необхідні подальші дослідження.

Таблиці з більш ніж трьома індексами і всі вони одно стовпчикові індекси

Використання *трьох* тут довільне; можна замінити його на *чотири* чи більше. Але справа в тому, що наявність лише одно стовпчикових індексів може бути ознакою досить наївного підходу до індексації, що заслуговує на дослідження.

Зрозуміло, що таблиці, до яких потрібно більше придивитись, це ті, які ви бачили, вони з'являються знову і знову в запитах і створюють значну напругу в системі.

Детальне дослідження

Крім індексів, які просто не потрібні, найпоширеніші помилки індексації стосуються композитних (складних) індексів. Композитні індекси є індексами на декількох стовпцях. Для створення індексу СУБД сканує таблицю, об'єднує значення стовпців, які потрібно індексувати, викликає результуючий «ключ», пов'язує з цим новим ключем фізичну адресу рядка (так як ви знаходите номер сторінки в книжковому індексі), сортує ключі, і будує на відсортованому результуючому списку дерево, що дозволяє швидко і легко досягти даного ключа. Загалом, це результат оператора create index.

Коли два критерії завжди використовуються разом, немає сенсу використовувати окремі індекси, навіть якщо обидва критерії є досить селективними: наявність двох індексів означає, що оптимізатор повинен вибрати, який індекс найкраще використовувати, або використовувати кожен окремо і об'єднати

результат, тобто повернути тільки ті рядки, які пов'язані з обома ключами. У будь-якому випадку, це буде більш складним, а отже, більш повільним, ніж просто пошук по одному індексу (підтримка індексу під час вставки та видалення також буде набагато менш дорогим процесом з меншою кількістю індексів).

Щоб порівняти композитні індекси з індексами з одним стовпцем, припустимо, що ми хочемо опитувати таблицю, яка записує англійські п'єси з кінця 16-го і початку 17-го століть. Ця таблиця містить таку інформацію, як назву п'єси, її жанр (комедія, трагедія, історія тощо), а також приблизний рік її створення. У стовпці жанру можуть міститися null значення, оскільки багато п'єс ніколи не друкувалися, були втрачені і відомі лише в бухгалтерських книгах (коли вони були оплачені) або в щоденниках сучасників; більше того, під час англійського ренесансу багато п'єс були ще середньовічними за природою і не були ні рибою, ні птицею, а просто влаштовували народні розповіді. Тепер, припустимо, ми хочемо знайти п'єси, які були ідентифіковані як комедії в нашій базі даних і які були створені між 1590 і 1595, і припустимо, що ми маємо індекс на жанр і індекс на рік створення.

Можна символічно представити записи в індексі жанру наступним чином:

	->	The Fancies Chaste and Noble	Ford
	->	A Challenge for Beautie	Heywood
	->	The Lady's Trial	Ford
comedy	**->**	**A Woman is a Weathercock**	**Field**
comedy	**->**	**Amends for Ladies**	**Field**
comedy	**->**	**Friar Bacon and Friar Bungay**	**Greene**
comedy	**->**	**The Jew of Malta**	**Marlowe**

comedy	**->**	**Mother Bombie**	**Lyly**
comedy	**->**	**The Two Gentlemen of Verona**	**Shakespeare**
comedy	**->**	**Summer's Last Will and Testament**	**Nashe**
... *many, many entries* ...			
comedy	**->**	**The Spanish Curate**	**Massinger,Fletcher**
comedy	**->**	**Rule a Wife and Have a Wife**	**Fletcher**
comedy	**->**	**The Elder Brother**	**Massinger,Fletcher**
comedy	**->**	**The Staple of News**	**Jonson**
comedy	**->**	**The New Inn**	**Jonson**
comedy	**->**	**The Magnetic Lady**	**Jonson**
comedy	**->**	**A Mayden-Head Well Lost**	**Heywood**
history	->	Richard III	Shakespeare
history	->	Edward II	Marlowe
history	->	Henry VI, part 1	Nashe,Shakespeare

....

Цей індекс вказує на 87 п’єс у таблиці, які ідентифікуються як комедії. Тим часом індекс на рік створення виглядає приблизно так:

...

1589	->	A Looking Glass for London and England	Greene
1589	->	Titus Andronicus	Shakespeare
1590	**->**	**Mother Bombie**	**Lyly**
1591	**->**	**The Two Gentlemen of Verona**	**Shakespeare**
1591	**->**	**Richard III**	**Shakespeare**
1592	**->**	**Summer's Last Will and Testament**	**Nashe**
1592	**->**	**Edward II**	**Marlowe**
1592	**->**	**Henry VI, part 1**	**Nashe,Shakespeare**
1592	**->**	**Henry VI, part 2**	**Nashe,Shakespeare**
1592	**->**	**Henry VI, part 3**	**Shakespeare,Nashe**
1593	**->**	**The Massacre at Paris**	**Marlowe**
1594	**->**	**The Comedy of Errors**	**Shakespeare**

1594	**->**	**The Taming of the Shrew**	**Shakespeare**
1594	**->**	**The History of Orlando Furioso**	**Greene**
1595	**->**	**A Midsummer Night's Dream**	**Shakespeare**
1595	**->**	**A Pleasant Conceited Comedy of George a Green**	**Greene**
1595	**->**	**Love's Labour's Lost**	**Shakespeare**
1595	**->**	**Richard II**	**Shakespeare**
1595	**->**	**The Tragedy of Romeo and Juliet**	**Shakespeare**
1596	->	A Tale of a Tub	Jonson

...

Тут у нас є лише 17 п'єс, які були створені між 1590 і 1595 рр., Але швидкого погляду на назви достатньо, щоб сказати нам, що деякі п'єси не комедії!

Що СУБД може зробити: прочитати (з індексу жанру) 87 посилань на комедії, окремо прочитати 17 посилань на п'єси, створені між 1590 і 1595, і зберегти тільки ті посилання, які є спільними для обох множин. Дуже грубо кажучи, "вартість" буде оцінюватись: прочитати 87 плюс 17 записів індексу, а потім сортувати їх, щоб знайти їх перетин.

Ми можемо порівняти цю вартість з вартістю доступу по індексу як за жанром, так і за роком створення. Оскільки рік створення набагато більш селективний, аніж жанр, то ми можемо спокуситися створити свій композитний індекс на (creation_year, жанр) у такому порядку:

...

1589|morality	->	A Looking Glass for London and England	Greene
1589|tragedy	->	Titus Andronicus	Shakespeare
1590|comedy	**->**	**Mother Bombie**	**Lyly**
1591|comedy	**->**	**The Two Gentlemen of Verona**	**Shakespeare**
1591|history	**->**	**Richard III**	**Shakespeare**
1592|comedy	**->**	**Summer's Last Will and Testament**	**Nashe**

1592|history -> Edward II Marlowe

1592|history -> Henry VI, part 1 Nashe,Shakespeare

1592|history -> Henry VI, part 2 Nashe,Shakespeare

1592|history -> Henry VI, part 3 Shakespeare,Nashe

1593|history -> The Massacre at Paris Marlowe

1594|comedy -> The Comedy of Errors Shakespeare

1594|comedy -> The Taming of the Shrew Shakespeare

1594|tragicomedy -> The History of Orlando Furioso Greene

1595|comedy -> A Midsummer Night's Dream Shakespeare

1595|comedy -> A Pleasant Conceited Comedy of George a Green Greene

1595|comedy -> Love's Labour's Lost Shakespeare

1595|history -> Richard II Shakespeare

1595|tragedy -> The Tragedy of Romeo and Juliet Shakespeare

1596|comedy -> A Tale of a Tub Jonson

1596|comedy -> The Merchant of Venice Shakespeare

...

Читаючи цей єдиний індекс, СУБД знає, що повернути і що відкинути. Проте саме в цьому запиті такий індекс не є найефективнішим, оскільки перший стовпець представляє, для всіх практичних цілей, основний ключ сортування, а другий - незначний ключ. Коли СУБД читає індекс, дерево переносить його на першу комедію, яку ми знаємо, створену в 1590 році, яка є John Lyly's *Mother Bombie*. Звідти він читає всі записи - комедії, трагедії, трагікомедії та історичні п'єси - до тих пір, поки він не потрапить у перший запис 1595 року, який не є комедією.

На відміну від цього, якщо ми індексуємо (жанр, creation_year), СУБД може почати, знову ж таки, з *Mother Bombie*, але може припинити читання індексів, коли вона зустрічає першу комедію 1596. Замість того, щоб читати 15 записів індексу, вона читає лише 8 записів, які дійсно вказують на рядки, які повинні бути повернуті з таблиці:

...

comedy|<null>	->	Amends for Ladies	Field
comedy|1589	->	Friar Bacon and Friar Bungay	Greene
comedy|1589	->	The Jew of Malta	Marlowe
comedy|1590	**->**	**Mother Bombie**	**Lyly**
comedy|1591	**->**	**The Two Gentlemen of Verona**	**Shakespeare**
comedy|1592	**->**	**Summer's Last Will and Testament**	**Nashe**
comedy|1594	**->**	**The Comedy of Errors**	**Shakespeare**
comedy|1594	**->**	**The Taming of the Shrew**	**Shakespeare**
comedy|1595	**->**	**A Midsummer Night's Dream**	**Shakespeare**
comedy|1595	**->**	**A Pleasant Conceited Comedy of George a Green**	**Greene**
comedy|1595	**->**	**Love's Labour's Lost**	**Shakespeare**
comedy|1596	->	A Tale of a Tub	Jonson
comedy|1596	->	The Merchant of Venice	Shakespeare
comedy|1597	->	An Humorous Day's Mirth	Chapman
comedy|1597	->	The Case is Altered	Jonson
comedy|1597	->	The Merry Wives of Windsor	Shakespeare
comedy|1597	->	The Two Angry Women of Abington	Porter

..

Звичайно, в цьому прикладі це не мало б різниці, але на деяких дуже великих таблицях, той факт, що один індекс йде прямо до потрібного місця, а інші довго думають, буде помітним. Коли умова визначає стан діапазону (як у create_year між 1590 і 1595) або просто нерівність у стовпчику, що належить до композитного індексу, пошуки швидше, коли цей стовпець з'являється в індексі після стовпців, на яких існує умова рівності (як у жанрі = 'комедія').

Погана новина полягає в тому, що ми не можемо робити все, що хочемо, з індексами. Якщо ми маємо індекс (жанр, creation_year) і часто зустрічається запит типу - "Які п'єси були створені в 1597 році?" Наш індекс буде сумнівним. СУБД може вибрати сканування індексу і перехід від жанру до жанру (включаючи невідомий або невизначений жанр), щоб знайти

діапазон записів з класу 1597. Або через складнішу структуру індексів СУБД може вважати, що дешевше (і тому швидше) і сканувати всю таблицю.

Оскільки цей тип даних не буде активно оновлюватися, має сенс мати, з одного боку, індекс на (жанр, creation_year) і з іншого боку індекс одного стовпчика на creation_year, щоб всі запити працювали швидко: запити, які включають жанр і запити, в яких його немає.

Основна проблема з композитними індексами - це завжди порядок стовпців. Припустимо, що ми маємо індекс на стовпцях C1, C2 і C3 у такому порядку. Маючи таку умову:

```
where C1 =
      value1
and C2 = value2
and C3 = value3
```

це майже так само, як просити вас знайти всі слова в словнику, чия перша літера - *i*, друга - *d*, а третя - *e* - ви переходите прямо до *idea* і читаєте всі слова до (але виключаючи) *idiocy*. Тепер припустимо, що умова така:

```
where C2 =
      value2
and C3 = value3
```

Еквівалентний словниковий пошук міг би знайти всі слова з *d* і *e* як другу і третю літери. Це набагато складніший у виконанні пошук, тому що тепер ми повинні перевірити всі розділи від *A* to *Z* і шукати відповідні слова.

В якості останнього прикладу припустимо, що ми маємо наступне:

```
where C1 =
      value1
```

```
and C3 = value3
```

Знаючи першу літеру, ми можемо зробити розумне використання індексу, але в цьому випадку ми повинні сканувати всі слова починаючи з *i*, перевіряючи третю букву, коли ми стикаємося з нею кожного разу, тоді як, якщо б нам дали першу і другу літери, наш пошук був би набагато ефективнішим.

Як можна бачити з композитними індексами це точно так само, як знання перших літер у слові для пошуку у словнику: якщо ви пропустите значення, яке відповідає першому стовпцю, ви не зможете зробити багато з індексом. Якщо вам надано значення першого стовпця, навіть якщо ви не знаєте значення для всіх стовпців, ви можете зробити досить ефективне використання вашого індексу - наскільки ефективним залежить від селективності стовпців, для яких є значення.

Поки стовпець не посилається на підфразу where, не можна використовувати стовпці, які слідують за ним у індексі, навіть якщо вони присутні в підфразі where з умовами рівності. Багато слабких застосувань індексів походять від неправильного впорядкування, де стовпці, які містять важливі критерії, "поховані" після стовпців, які рідко з'являються в пошуках.

Переглядаючи індекси, слід перевірити ряд речей:

- Одно стовпчикові індекси, які також з'являються на першій позиції композитного індексу, є надлишковими, оскільки можна використовувати композитний індекс, можливо, трохи менш ефективно, але все ж ефективно. Часто можна знайти надлишкове індексування у випадку, наприклад, класичний набір студентів: один студент може відвідувати декілька курсів, а кілька студентів відвідують один курс. Таким чином, таблиця реєстрації буде

пов'язувати studentid to a courseid; первинним ключем для цієї таблиці буде studentid, плюс courseid, плюс, ймовірно, семестр. Але studentid та courseid є також foreign ключі, відповідно вказуючи на таблицю students і таблицю courses. Якщо ви систематично індексуєте всі foreign ключі, то ви створите індекс на courseid (fine) і індекс на studentid (зайвий стосовно первинного ключа, який починається з цього стовпця). Під час запиту не буде втрат ефективності, але будуть втрати при вставці.

- Для створення індексів (особливо, якщо вони не визначені як unique!), які є супермножинами первинного ключа, слід мати чітке обґрунтування (не є обґрунтуванням те, що його не визначили як unique). Іноді такі індекси створюються таким чином, щоб знайти всі дані, необхідні для запиту по індексу, і заощадити доступ до самої таблиці.
- Стовпці, які *завжди* з’являються одночасно у фразі where, повинні бути індексовані композитним індексом, а не окремими одно стовпчиковими індексами.
- Коли стовпці *часто* з'являються разом, то стовпці, які можуть з'явитися без інших, повинні з'явитися першими у композитному індексі, за умови, що вони достатньо селективні, щоб зробити пошук по індексу більш ефективним, ніж повне сканування. Якщо всі стовпці можуть з'являтися окремо і є досить селективними, то має сенс окремо індексувати деякі стовпці, які не відображаються на першій позиції у композитному індексі.
- Слід перевірити, що всякий раз, коли існує умова діапазону на стовпці, яка завжди або дуже часто асоціюється з умовами рівності в інших стовпцях, то ці

інші стовпці мають з'являтися першими в композитному індексі.

Індекси B-дерев на стовпцях з малою кількістю елементів безглузді, якщо значення рівномірно розподілені, або якщо немає гістограми значення (оскільки оптимізатор припускає, що значення рівномірно розподілені). Наприклад, якщо у таблиці students є колонка gender, то, як правило, немає сенсу індексувати її, тому що можна очікувати приблизно стільки ж чоловіків, скільки студенток у вашій популяції; ви ж не хочете використовувати індекс для отримання 50% рядків. Проте, обґрунтування може бути іншим у військовій академії, де студентки представляють невеликий відсоток від усіх курсантів, і якщо жінка є селективним критерієм. Але потім для СУБД потрібна гістограма величин.

Індекси, які порушують правила

Як завжди, існують винятки із загальних рекомендацій, головним чином через особливі типи індексів.

Bitmap indexes

Наприклад, Oracle дозволяє bitmap індекси, які спеціально розроблені для систем підтримки прийняття рішень. Індекси bitmap корисні, в першу чергу, для даних, які можуть масово вставлятися, але не (або рідко) оновлюватися (їх слабкий момент полягає в тому, що вони погано керують конфліктами при паралельних сесіях), і по-друге, для стовпців з низьким числом різних значень. Ідея bitmap індексів полягає в тому, що якщо у вас є декілька умов на кількох колонках з низькою потужністю, то комбінування bitmap через логічні OR та AND дозволяє скоротити кількість рядків-кандидатів дуже швидко та дуже ефективно. Хоча реалізація і відрізняється, але основний принцип bitmap індексів ближче до операцій, які можна виконувати з full-text індексами, ніж до регулярних B-tree індексів, за винятком того, що, як і у регулярних B-tree індексів, справедливо наступне:

- Bitmap індекси дійсно індексують значення стовпців, а не частини стовпця, як full-text індекси.
- Bitmap індекси повністю беруть участь у реляційних операціях, таких як з'єднання.
- Bitmap індекси підтримуються (з раніше згаданими обмеженнями) в реальному часі.

Якщо ви коли-небудь зустрінете bitmap індекси, дотримуйтесь таких правил:

- Якщо система з транзакціями, не використовуйте bitmap індекси на сильно оновлюваних таблицях.

- Не ізолюйте bitmap індекси. Сила bitmap індексів знаходиться в тому, як ви їх комбінуєте. Якщо у вас є один bitmap індекс серед декількох регулярних B-tree індексів, то оптимізатор матиме проблеми з ефективним використанням цього самітнього bitmap індексу.

Кластерні індекси

Ви знайдете інший основний тип виключення в кластерних індексах, про що вже згадувалось. Кластерні індекси змушують рядки таблиці зберігатися в тому ж порядку, що й ключі в індексі. Кластерні індекси є основною функцією як SQL Server, так і MySQL (з механізмом зберігання, таким як InnoDB), і вони мають близьких родичів в індекс-організованих таблицях Oracle. В кластерних індексах ключі відносять вас безпосередньо до даних, а не до адрес даних, оскільки таблиця та індекс об'єднуються, що також економить пам'ять. Оскільки оновлення ключа означатиме фізичне переміщення цілого рядка для його збереження на новому місці, кластерний індекс (очевидно, може бути лише один на таблицю) зазвичай є первинним ключем (не кажіть про оновлення первинного ключа) або квазіключем – тобто, комбінацією обов'язкових та unique стовпців.

Є одна річ для перевірки кластерних індексів: чи є порядок рядків відповідним тому, що ви хочете зробити? Три переваги кластеризованих індексів полягають у тому, що ви переноситесь безпосередньо до рядків, рядки групуються разом (як випливає з назви) для суміжних значень стовпців (ключів) ключа, і вони перенаправлені, що може зберегти СУБД багато роботи в наступних сценаріях:

- При використанні фраз order by
- При використанні фраз group by
- При використанні функцій ранжування таких як rank( ) або row_number( )

Це також може виявитися корисним при з'єднаннях. Якщо порівнювані стовпці знаходяться в одному і тому ж порядку в двох таблицях, що може статися, якщо первинний ключ першої таблиці є першим стовпцем у складеному первинному ключі другої таблиці, знаходження відповідностей здійснюється легко.

Також зауважимо, що якщо ви систематично використовуєте в якості первинного ключа семантично ненавантажений стовпець (стовпець автоінкремента для користувачів MySQL), що не використовується як ключ сортування, а також не є критерієм для групування або з'єднання, то ваша кластерна таблиця не має переваги над звичайною таблицею.

Indexes on expressions

Нарешті трохи про індекси на обчислюваних стовпцях (іноді відомих як *function-based indexes*). Як вказувалось у прикладі пошуку по словнику, як тільки ви шукаєте не слово, а трансформацію слова (наприклад, слова, що містять *a* в третій позиції), єдине використання словника буде як список слів, який вам доведеться сканувати, без будь-якої надії зробити це ефективно. Якщо ви застосуєте функцію до стовпця і порівняйте його з константою, ви зробите індекс на цій колонці непридатним, тому замість цього вам слід застосувати обернену функцію до константи. Однак існує обмеження: індексований

вираз повинен бути детермінованим, тобто він повинен завжди повертати одне і те ж значення, коли він подається з однаковими аргументами.

Синтаксичний аналіз та змінні зв’язку

Окрім статистики та індексування (це незалежно від коду програми) один аспект, який безпосередньо стосується коду програми, також повинен перевірятися на дуже ранній стадії: оператори hardcoded (жорсткі) чи softcoded. *Hardcoded statements* є операторами, в яких всі константи є невід'ємною частиною оператора; *softcoded statements* є операторами, в яких константи з високим ступенем мінливості між послідовними виконаннями передаються як аргументи. Раніше вказувалось, що hardcoded оператори ускладнюють отримання справжнього уявлення про завантаження сервера СУБД, і тому вони можуть змусити нас зосередитися на неправильних напрямках, якщо ми недостатньо обережні.

Вони також можуть завдати серйозних проблем (особливо вірно для Oracle). Щоб зрозуміти, що саме завантажує сервер, ми повинні зрозуміти, що відбувається на кожному етапі, і хорошим місцем для початку є опис того, що відбувається, коли ми видаємо оператор SQL. З точки зору розробника ми записуємо оператор SQL до змінної рядка символів. Цей оператор може бути константою або може бути динамічно побудований програмою. Потім ми викликаємо функцію, яка виконує оператор. Ми перевіряємо код помилки, і якщо немає помилки, а

оператор був select, ми в циклі вибираємо (fetch) рядки, використовуючи іншу функцію, поки вона не поверне нічого.

Але, що відбувається на стороні сервера?

SQL - це мова, що спеціалізується на зберіганні, виборі та модифікації даних. Як і всі мови, вона має бути трансльована в машинний код або, принаймні, в деякий проміжний, базовий код. Однак, трансляція є важчою, ніж у більшості мов. Розглянемо трансляцію синонімів, контроль прав доступу до таблиць, інтерпретацію * в операторі select * і т.д. Всі ці операції вимагатимуть застосування рекурсивних запитів до словника даних. Ще гірше, врахуйте роль, яку відіграє оптимізатор, і вибір найкращого плану виконання: ідентифікацію індексів, перевірка того, чи вони корисні, обчислення того, в якому порядку відвідувати численні таблиці, на які посилається складне з'єднання, включення переглядів у вже складний запит тощо. Парсінг SQL-запиту є дуже складною і, отже, вартісною операцією. Чим менше ви це робите, тим краще.

Як визначити проблеми синтаксичного аналізу

Існує простіший спосіб перевірити проблеми парсінгу, ніж спроба проаналізувати те, що виконується, за умови, що у вас є права на запит правильних системних views. В Oracle v$ssystat розповість вам все, що потрібно знати, в тому числі, скільки CPU часу використовувала СУБД, а парсінг використовував його після запуску:

```
select sum(case name

                when 'execute count' then value
```

```
        else 0
      end) executions,
sum(case name
        when 'parse count (hard)' then value
        else 0
         end) hard_parse,
           round(100 * sum(case name
                             when 'parse count (hard)' then value
                             else 0
                           end) /
                       sum(case name
                             when 'execute count' then value
                             else 0
                           end), 1) pct_hardcoded,
           round(100 * sum(case name
                             when 'parse time cpu' then value
                             else 0
                           end) /
                       sum(case name
                             when 'CPU used by this session' then value
                             else 0
                           end), 1) pct_cpu
```

```
from v$sysstat

where name in ('parse time cpu',

                'parse count (hard)',

                'CPU used by this session',

                'execute count');
```

MySQL- information_schema.global_status надасть вам подібну інформацію, але без часу для CPU:

```
select x.queries_and_dml,

       x.queries_and_dml - x.executed_prepared_stmt
hard_coded, x.prepared_stmt,

       round(100 * (x.queries_and_dml - x.executed_prepared_stmt)

                   x.queries_and_dml,1) pct_hardcoded
from (select sum(case variable_name

                   when 'COM_STMT_EXECUTE' then 0

                   when 'COM_STMT_PREPARE' then 0

                   else cast(variable_value as unsigned)

               end) as queries_and_dml,

           sum(case variable_name

                   when 'COM_STMT_PREPARE' then cast(variable_value as
                   unsigned)

                   else 0

               end) as prepared_stmt,

           sum(case variable_name
```

```
                when 'COM_STMT_EXECUTE' then cast(variable_value as
                unsigned)

                else 0

            end) as executed_prepared_stmt

from information_schema.global_status

where variable_name in ('COM_INSERT',

                        'COM_DELETE',

                        'COM_DELETE_MULTI',

                        'COM_INSERT_SELECT',

                        'COM_REPLACE',

                        'COM_REPLACE_SELECT',

                        'COM_SELECT',

                        'COM_STMT_EXECUTE',

                        'COM_STMT_PREPARE',

                        'COM_UPDATE',

                        'COM_UPDATE_MULTI')) x;
```

Зверніть увагу, що оскільки всі значення є кумулятивними, то якщо ви хочете керувати певною частиною програми, ви повинні взяти snapshots до і після і перерахувати співвідношення.

SQL Server надає безпосередньо деякі лічильники частоти парсінгу. Щоб дати вам уявлення про цифри, які ви повинні очікувати в корпоративному середовищі слід завжди дивитися в Oracle відношення парсінгу до виконання в районі від 3% до 4%,

коли цей аспект кодування був задовільним. Немає жодних сумнівів з 8% або 9%, але, звичайно, справжнє питання полягає в тому, скільки цей парсінг коштує нам (хоча індикатор CPU, наданий Oracle, є дуже хорошим показником), і, за допомогою виведення того, на який виграш можна сподіватися, виправивши код.

Оцінка втрат продуктивності із-за синтаксичного аналізу

Щоб отримати оцінку того, скільки ми можемо втратити на парсінгу, можна провести дуже простий тест на Oracle, SQL Server і MySQL. Більш конкретно, потрібно запустити на одній з таблиць (з попереднього прикладу) JDBC програму *HardCoded.java*, найважливішою частиною якої є наступний фрагмент:

```
start = System.currentTimeMillis( );
try {
    long    txid;
    float   amount;
    Statement st = con.createStatement( );
    ResultSet rs;
    String  txt;

    for (txid = 1000; txid <= 101000; txid++){
       txt = "select amount"
```

```
                " from transactions"

                " where txid=" +
            Long.toString(txid); rs =
            st.executeQuery(txt);

            if (rs.next( )) {

                amount = rs.getFloat(1);

            }

        }

        rs.close( );

        st.close( );

    } catch(SQLException ex){
            System.err.println("==>
            SQLException: "); while (ex != null) {


                System.out.println("Message:   " + ex.getMessage ( ));

                System.out.println("SQLState:  " + ex.getSQLState ( ));

                System.out.println("ErrorCode: " + ex.getErrorCode ( ));

                ex = ex.getNextException( );

                System.out.println("");

            }

    }

    stop = System.currentTimeMillis( );
    System.out.println("HardCoded - Elapsed (ms)\t" + (stop -
    start));
```

Ця програма виконує 100,000 разів у циклі простий запит, який отримує значення з єдиного рядка, пов'язаного з значенням первинного ключа; Ви помітите, що індекс

циклу конвертується в string, а потім він просто конкатенується до тексту запиту перед тим, як оператор передається на DBMS сервер на виконання.

Якщо ви повторно виконуєте запит, то є сенс написати його у вигляді функції, тобто шматок коду, який приймає параметри і виконується повторно з різними аргументами (тут немає нічого особливого для SQL). Всякий раз, коли ви виконуєте подібну операцію з константами, які змінюються з кожним виконанням, ви не повинні " hardcode " ці константи, а перейти в три етапи:

1. Підготувати оператор з маркерами місця для констант.

2. Прив’язати значення до операторів—тобто, забезпечити вказівники на змінні, які під час виконання замінять маркери місця (назва *bind variable* зазвичай використовується в Oracle).

3. Виконати оператор.

Коли потрібно повторно виконати оператор з іншими значеннями, то все, що потрібно зробити, це приписати нові значення змінним і повторити 3-ій крок.

Тому *HardCoded.java* було переписано з використанням prepared statement замість варіанта динамічно побудованого hardcoded оператора, але двома способами:

- У варіанті *FirmCoded.java* в кожній ітерації циклу створюється новий prepared statement with a placeholder for txid, який зв’язує поточне значення txid з оператором, який

виконується, вибирається значення amount, і оператор закривається перед інкрементуванням txid.

- У варіанті *SoftCoded.java* prepared statement створюється один раз, і просто змінюється значення параметра перед повторним виконанням оператора і вибором значення amount в циклі.

На Рис.1 зображено відносні результати виконання *FirmCoded* та *SoftCoded*, порівнювані з *HardCoded* для кожної СУБД із заданого набору.

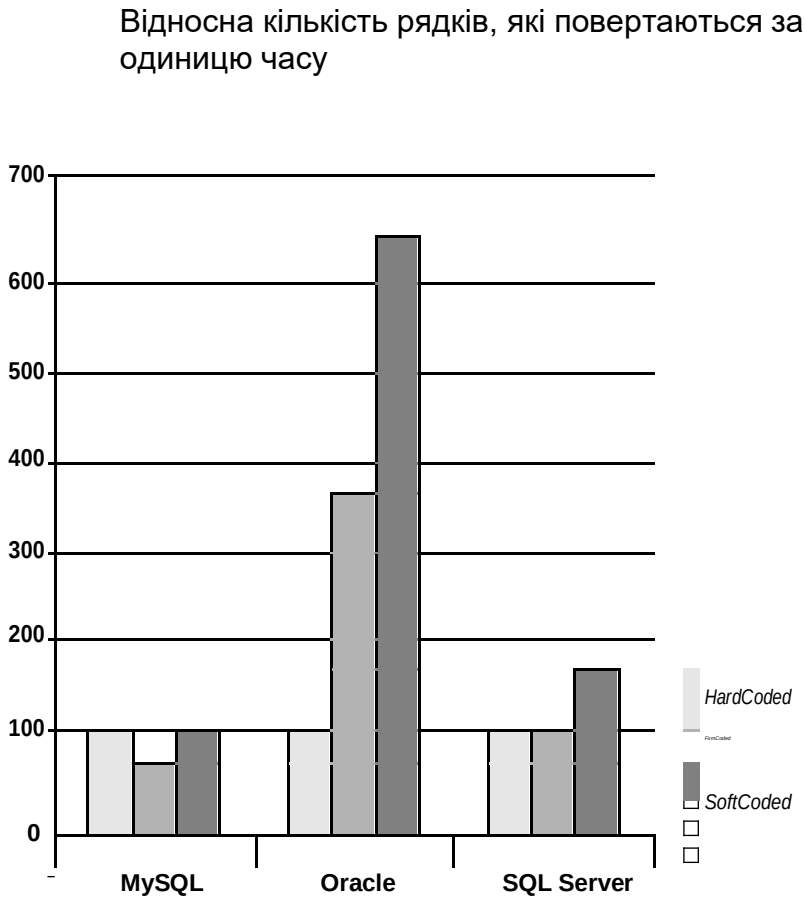

Рис 1 . Втрати продуктивності

Відмінності дуже цікаві, особливо у випадку з *FirmCoded.java*: для Oracle є великий приріст продуктивності, тоді як для MySQL є помітне зниження продуктивності. Чому так? Дійсно, документація MySQL Connector/J (драйвер JDBC) стверджує, що prepared statements реалізовані драйвером, і одна і в тій ж документації відмічається різниця між prepared statements на стороні клієнта і реалізованими на стороні сервера. Насправді, це не питання мови або драйвера: якщо переписати всі три програми на C, і хоча все працює швидше, ніж в Java, співвідношення продуктивності буде суворо ідентичне серед *HardCoded, FirmCoded і SoftCoded*. Різниця між MySQL і Oracle

полягає в тому, як сервер кешує оператори. Результати запиту в MySQL кешуються: якщо однакові запити, що повертають кілька рядків, виконуються часто (що типово є шаблоном, який ми бачимо на популярному веб-сайті, який використовує систему управління вмістом на основі MySQL або CMS(content management system)), запит виконується один раз, його результат кешується, і поки таблиця не змінюється, результат пізніше вибирається з кешу кожен раз, коли один і той же запит передається серверу. Oracle також має кеш запитів, хоча він і був введений із запізненням (з Oracle 11g); однак протягом десятиліть він використовував кеш операторів, де зберігаються плани виконання, а не результати. Якщо подібний (байт в байт) запит вже виконано на тих самих таблицях, і він все ще знаходиться в кеші, то він повторно використовується навіть іншою сесією, якщо середовище ідентичне. (Сесія може локально налаштувати деякі параметри, які впливають на спосіб виконання оператора.)

В Oracle, кожен раз, коли новий запит передається на сервер, запит перевіряється на контрольну суму(КС), середовище контролюється, і по кешу виконується пошук; якщо знайдено подібний запит, то ми просто маємо цей "soft parse", і запит виконується. В іншому випадку відбувається справжня операція парсінгу, і ми маємо дорогий «hard parse».

При *FirmCoded* ми замінюємо hard parses через soft parses, а при *SoftCoded* ми повністю позбавляємося від soft parses. Якщо запускаються різноманітні програми через команду time Unix, як це зроблено на рис. 2, то видно, що різниця в загальному часі відбувається повністю на стороні сервера, оскільки використання CPU на стороні клієнта (the sum of sys and user)

залишається більш-менш однаковим. Насправді, оскільки *FirmCoded* виконує багаторазове виділення та звільнення prepared statement об'єкта, то цей варіант програми споживає більшу частину ресурсів на стороні клієнта.

На відміну від цього, для MySQL, коли ми prepare оператор, ми асоціюємо його з певним планом, але якщо кеш запитів є активним (це не за замовчуванням), то обчислюється контрольна сума після заміни параметрів, щоб побачити, чи може результат бути взятим з кешу. І до тих пір, поки зв'язані змінні будуть різними, “prepared” оператор аналізується(hard-parsed). Як наслідок, в *FirmCoded*, де оператори готуються(prepared) до виконання лише один раз, немає жодної вигоди. Ми чітко бачимо втрату ефективності, пов'язану з більш вартісною стороною клієнта. Як тільки оператор виконується з різними значеннями, як у випадку з *SoftCoded*, зв'язок між оператором і його планом виконання зберігається, і є значні переваги продуктивності, хоч і не такі вражаючі, як у Oracle.

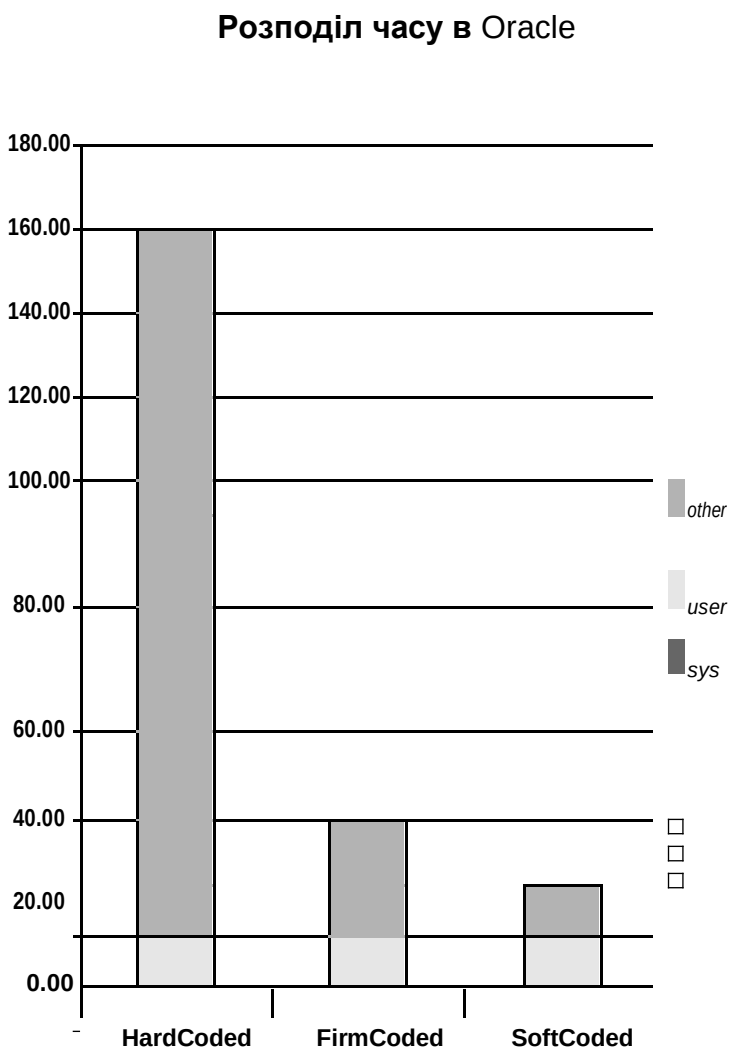

Рис. 2 Команда *Unix time* про витрачений час

SQL Server знаходиться десь посередині, і навіть якщо на основі результатів він виглядає ближче до MySQL, ніж до Oracle, його внутрішні механізми насправді набагато ближче до Oracle (плани виконання можуть розподілені). Розмиває картину те, що за простим запитом, як у наведеному прикладі, SQL Server виконує *просту параметризацію* hardcoded оператора - коли він аналізує hardcoded оператор, і якщо він виглядає досить безпечним (з первинним ключем, це досить безпечно)), СУБД просто відриває константу і передає її як параметр. У випадку з SQL Server

hardcoded операторі через внутрішні механізми СУБД були перетворені в щось подібне до prepared операторів *FirmCoded.java*; як наслідок, початкова ефективність була насправді набагато кращою ніж у Oracle. *FirmCoded.java* була трохи кращою, тому що кожен раз, коли ми готували(prepared) оператор, ми зберігали цю додаткову обробку видобування константних значень перед тим, як перевірити, чи виконувався цей оператор раніше. Слід підкреслити, однак, що проста параметризація не виконується агресивно, а навпаки, і тому вона «працює» тільки у простих випадках.

Перший (і проміжний) висновок: hardcoding оператори межують з самогубством на Oracle, їх слід уникати на SQL Server, і завдає серйозних втрат у продуктивності на будь-якій СУБД, коли одна сесія повторно запускає оператори з однаковим шаблоном.

Так чи інакше, використання prepared операторів на Oracle і на SQL Server є альтруїстичним: якщо всі сесії використовують prepared оператори, то всім вигідно, тому що тільки одна сесія пройде через жорсткий розбір(parse). На відміну від цього на MySQL це більш егоїстична справа: одна сесія значно виграє (50% у згаданому прикладі), якщо вона повторно виконує той самий запит, але інші сесії не виграють. Слід пам'ятати, що тут говориться про сесії бази даних; якщо ви використовуєте сервер прикладних програм, який об'єднує декілька користувацьких сесій, і ви змушуєте їх розподіляти менше число сесій баз даних, то сесії користувачів можуть отримати взаємну вигоду.

Корекція проблем синтаксичного аналізу

Навіть розробники з певним досвідом часто починають використовувати hardcoding оператори, коли їхні оператори динамічно генеруються «на льоту» - це виглядає так природно конкатенувати значення після конкатенації фрагментів коду SQL. Але тут є підводні камені.

По-перше, при конкатенації значень, і особливо рядків з веб-форм застосувань, якщо не бути особливо обережним, то можливий великий ризик (якщо ви ніколи не чули, що SQL пов'язаний з *injection*, то саме час використати свою улюблену пошукову систему веб-пошуку).

По-друге, схоже багато розробників вважають, що hardcoding оператор не має великого значення, коли «все настільки динамічно», тим більше, що є практичні обмеження щодо кількості prepared операторів. Але навіть коли користувачі можуть вибирати будь-яку кількість критеріїв серед численних можливостей, кількість комбінацій може бути великою (вона піднята до рівня степені 2 від кількості критеріїв), але широкі масиви можливих комбінацій зазвичай посилюються тим, що деякі комбінації надзвичайно популярні, а інші дуже рідко використовуються. Як наслідок, реальне розмаїття залежить від значень, які вводяться і асоціюються з критерієм, набагато більше, ніж від вибору критеріїв.

Перехід від hardcoded до "softcoded" операторів зазвичай не вимагає великих зусиль. При використанні prepared оператора попередній фрагмент коду для hardcoded оператора виглядає наступним чином:

```
start = System.currentTimeMillis( );

try {

      long       txid;

      float      amount;

PreparedStatement st = con.prepareStatement("select amount"

                                                  + " from transactions"

                                                  + " where txid=?");

      ResultSet rs;


      for (txid = 1000; txid <= 101000; txid++){

          st.setLong(1, txid);

          rs = st.executeQuery( );

          if (rs.next( )) {

              amount = rs.getFloat(1);

          }

      }

      rs.close( );

      st.close( );

} catch(SQLException ex){
       System.err.println("==>
       SQLException: "); while (ex != null) {


            System.out.println("Message:   " + ex.getMessage ( ));

            System.out.println("SQLState:  " + ex.getSQLState ( ));
```

```
            System.out.println("ErrorCode: " +
            ex.getErrorCode ( )); ex = ex.getNextException(
            ); System.out.println("");


        }

    }

    stop = System.currentTimeMillis( );
    System.out.println("SoftCoded - Elapsed (ms)\t" + (stop -
    start));
```

Модифікація коду дуже проста, коли (як і в цьому випадку) число значень для "параметризації" є постійним. Проблема, однак, полягає в тому, що, коли оператори є динамічно побудованими, то зазвичай кількість умов може змінюватись, і, як наслідок, кількість параметрів змінюється теж. Було б дуже зручно, якби можна було б з'єднати фрагмент SQL-коду (з маркером місця) з відповідною змінною. На жаль, так воно не працює (і могли б бути труднощі при повторному виконанні оператора): спочатку готується оператор, а потім зв'язані з ним значення, але не можна підготувати оператор до того, як він буде повністю побудований.

Однак, оскільки побудова оператора, як правило, керується полями, введеними користувачем, то не дуже складно двічі пройти список полів - один раз, щоб створити оператор із заповнювачами і ще один раз для прив’язки фактичних значень до заповнювачів.

Що робити, якщо одне значення має бути зв’язане кілька разів?

Деякі мови дозволяють використовувати іменований маркер (наприклад, якщо ви кодуєте в C # або в Visual Basic, ви можете викликати маркер місця @name). Якщо ви кілька разів посилаєтесь на змінну у своєму операторі, вам потрібно, в .NET контексті викликати метод

SqlCommand Parameters.AddWithValue ("@name ", ...) лише один раз, щоб пов’язати одне і те ж значення з усіма входженнями @name в операторі.

Деякі мови, однак, дозволяють використовувати лише загальний маркер місця (наприклад,?), і це позиція маркерів місць, які підраховуються. Коли кожен параметр з’являється лише один раз, то не має великої різниці з іменованими маркерами місць. Але коли вам потрібно посилатися на один і той же параметр кілька разів, то його прив'язування може стати незграбною справою. Наприклад, якщо ви хочете отримати значення станом на певну дату, ви можете створити оператор, який виглядає приблизно так:

```
select whatever
from list of tables
where ...
    and effective_date <= ?
    and until_date > ?
 ...
```

де ? заповнювач в обох випадках повинен приймати однакове значення.

Часто ви можете легко подолати таку складність, трохи обдуривши і всунувши всередину запиту фіктивний підзапит, який дозволяє вам звернутися до того, що він повертає. Таким чином, на Oracle:

```
select whatever

from list of tables

     (select ? as val from dual) as
mydate where ...

   and effective_date <=
   mydate.val and
   until_date > mydate.val
```

...Заміна hardcoded операторів softcoded операторами - це не є величезним зусиллям щодо перезапису, але і не те, що можна робити механічно, тому люди часто спокушаються найпростішим рішенням - дозволити СУБД виконувати цю роботу.

Ледачий шлях корекції проблем синтаксичного аналізу

Ви бачили, що SQL Server намагається самостійно виправити проблеми аналізу. Ця спроба несмілива, і ви можете зробити її набагато сміливішою, встановивши параметр PARAMETERIZATION бази даних на FORCED. Як ви могли очікувати, в Oracle існує еквівалентна функція з параметром CURSOR_SHARING, значенням за замовчуванням якого є EXACT (тобто повторне використання планів виконання лише тоді, коли оператори є абсолютно однаковими), але їх можна встановити на SIMILAR (що дуже близьке до значення за замовчуванням SIMPLE для параметра PARAMETERIZATION для SQL Server) або до FORCE. Однак існує ряд незначних відмінностей; наприклад, параметр може бути встановлений Oracle або на всю б/д, або в межах сеансу. Oracle також поводиться дещо агресивніше при такому запиті ніж SQL Server.

Наприклад, SQL Server залишив би такий фрагмент коду без змін:

```
and char_column like 'pattern%'
```

а Oracle перетворив би його в таке:

```
and char_column like :"SYS_B_n" || '%'
```

Аналогічно, SQL Server залишить оператор, який вже містить принаймні один параметр, як є, тоді як Oracle параметризує його далі. Але за великим рахунком, ідея, що стоїть за цією функцією, однакова: змусити СУБД робити те, що в першу чергу повинні були б зробити розробники.

Повторний запуск своїх тестів з Oracle, виконавши (з привілеями адміністратора б/д) такої команди:

```
alter system set cursor_sharing=similar;
```

Як вже зазначалось, цей параметр дає інструкції Oracle вести себе так, як робить SQL Server, не вимагаючи цього. І якщо я заміню цифри, отримані раніше в Oracle, новим результатом, я отримую коефіцієнти поліпшення, які набагато більше відповідають тому, що ми отримали з SQL Server, тому що і для SQL Server, і для Oracle, якщо оператори все ще hardcoded в програмі в *HardCoded.java*, під час їх виконання вони більше не hardcoded (див. Рис.3).

З практичної точки зору, якщо ми працюємо на Oracle і помічаємо проблеми синтаксичного аналізу, однією з наших перших перевірок правильності повинно бути перевірка налаштування параметра cursor_sharing і, якщо буде exact, то

встановити його на similar та вивчити, чи покращує це продуктивність.

Не забувайте, що база даних не має на увазі одне й те те саме, що у світі SQL Server та MySQL, (розмова про сервер).

Відносна кількість рядків, повернених за одиницю часу з CURSOR_SHARING, встановленому на SIMILAR для Oracle

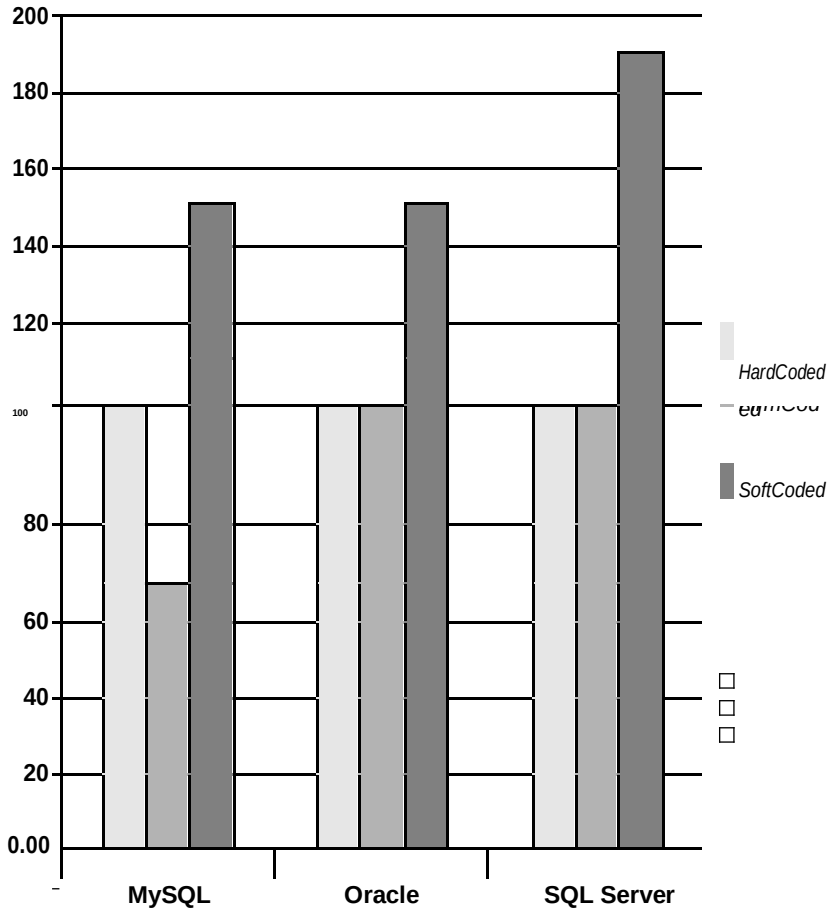

Рис. 3 . Зниження продуктивності з увімкненим фонового пом'якшення

Корекція належним чином проблем синт. аналізу

Навіщо турбуватися використанням підготовлених операторів, якщо СУБД може це зробити для нас? По-перше, ми можемо побачити на прикладах як Oracle, так і SQL Server, що якщо автоматизований процес параметризації є лише дещо менш ефективним, ніж prepared оператори, то оператори, які готуються

один раз і виконуються багато разів, принаймні в 1,5 рази швидші, ніж оператори, які параметризуються перед кожним виконанням. По-друге, наявність функції «AutoCorrect» під опцією «Tools» на панелі меню у вашому текстовому процесорі не означає, що вам слід ігнорувати граматику і покладатися на автоматичне виправлення. По-третє, менш агресивний тип системної параметризації помиляється на стороні обережності і за допомогою реальних прикл. програм полегшить лише частину болю; більш агресивний тип може закинути вас у глибокі неприємності.

Треба бути обережними і не замінювати кожну константу маркером місця. Якщо у вас стовпець із низькою кардинальністю (тобто невелика кількість різних значень), передача значень під час виконання може насправді бути шкідливою, якщо значення не розподіляються рівномірно; простою англійською мовою ви можете мати неприємні сюрпризи, якщо одне значення є дуже поширеним, а інші - порівняно рідкісними (звичайне явище зі стовпцями, що фіксують стан). Болісний випадок, це коли стовпець з кількома значеннями індексується, оскільки рідкісні значення надзвичайно вибіркові. Коли оптимізатор вирішує, яким буде найкращий план виконання, і якщо під час сканування оператора він зіткнеться з чимось таким, як:

```
and status = ?
```

і якщо стовпець статусу індексується, то оптимізатор повинен вирішити, чи слід використовувати цей індекс (можливо, надаючи йому перевагу перед іншим індексом). При операції join це може впливати на рішення відвідування таблиці, до якої status належить перш за все. Завжди метою оптимізатора є виділити те,

що стане набором результатів якомога швидше, а це означає, що треба відфільтрувати рядки, які не можуть належати до набору результатів якомога раніше.

Проблема з маркерами місць полягає в тому, що вони не дають поняття про фактичне значення, яке буде передано під час виконання. На цьому етапі існує кілька можливостей, які залежать від того, що оптимізатор знає про дані (статистичні дані, які зазвичай регулярно збираються в рамках адміністрування баз даних):

1. Статистика ніколи не збиралася, і оптимізатор взагалі нічого не знає про те, що містить стовпець. У цьому випадку єдине, що він знає, - це те, що індекс не є унікальним. Якщо оптимізатор має вибір між індексом неунікальним та унікальним індексом, він віддасть перевагу унікальному індексу, оскільки одне ключове значення може, максимум, повернути один рядок з унікальним індексом. Якщо у оптимізатора є вибір лише між неунікальними індексами, то це залежить від вашої удачі. Отже, важливість статистики.
2. Статистика не надто точна, і все, що оптимізатор знає, це те, що стовпець містить кілька різних значень. У такому випадку він, як правило, ігнорує індекс, що може бути досить невдалим, коли умова щодо status є найбільш селективною.
3. Статистика є точною, і оптимізатор також знає, використовуючи частотні гістограми, що одні значення дуже селективні, а інші зовсім не селективні. При порівннні із маркером місця оптимізатору доведеться вибрати з наступного:

- - Нестримний оптимізм, або припускаючи, що передане значення завжди буде дуже селективним, а отже, надавати перевагу плану виконання, який надає перевагу умові щодо status. Незважаючи на приємну алітерацію, ви навряд чи зустрінете оптимістичні оптимізатори, адже оптимізм може дати відсіч.
- - Грати в безпеці, це означає ігнорувати індекс, оскільки статистично це, ймовірно, може бути сумнівним. Ми повернулися до того випадку, коли оптимізатор знає лише те, що є мало різних значень і більш нічого, це може бути неправильною тактикою, коли умова про status буває найселективнішою в запиті.
- - Бути розумним, означає нерішуче зазирнути на значення, яке передається вперше (це відомо як зв'язана змінна, яка аналізується Oracle *bind variable peeking*). Якщо значення селективне, то буде використаний індекс; якщо ні, то індекс буде проігноровано. Це все дуже добре, але що робити, якщо вранці ми використовуємо селективні значення, а потім шаблон запитів змінюється протягом дня, і ми закінчуємо запити згодом з найбільш поширеним значенням? Або що робити, якщо відбувається зворотний рух (reverse)? Що робити, якщо з якоїсь причини повторно буде проаналізований запит із значеннями, що призводять до абсолютно іншого плану виконання, який буде застосовано для великої кількості виконань? У цьому випадку

користувачі стануть свідками випадкової поведінки запитів - іноді швидкими, а іноді повільними. Така поведінка запитів рідко буде популярною у кінцевих користувачів.

- - Бути ще розумнішим і знаючи, що може виникнути проблема з одним стовпцем, і перевіряючи це значення стовпця щоразу (на практиці це означає, що він поводиться так, ніби це значення було hardcoded).

Маркери місця є великою проблемою для оптимізаторів, коли вони замінюють значення, які порівнюються з індексованими стовпцями, що містять невелику кількість нерівномірно розподілених значень. Іноді оптимізатори роблять правильний вибір, але дуже часто, навіть коли вони намагаються бути розумними, вони роблять неправильну ставку, або вони вибирають, здавалося б, неправильний спосіб, який у деяких випадках може виділятися як дуже поганий.

Для стовпців з низькою кардинальністю, що зберігають нерівномірно розподілені значення, найкращий спосіб кодування, якщо розподіл даних досить статичний і немає ризику SQL-ін’єкції, – це жорстке кодування значень, з якими вони порівнюються у фразі where. І навпаки, всі рівномірно розподілені значення, які змінюються між послідовними виконаннями, повинні бути передані як аргументи.

Обробка списків в Prepared операторах

На жаль, є один випадок, який трапляється часто (зокрема, у динамічно генерованих операторах) і його особливо складно обробляти, коли ви серйозно ставитеся до належної підготовки операторів: фраза in (...), де ... є списком, розділеним комами, а не підзапитом.

У деяких випадках, фраза in проблем не створює:

- Коли ми перевіряємо стовпець status або будь-який інший стовпець, який може приймати одне з кількох можливих значень, пропозиція in повинна залишатися hardcoded з тієї самої причини, яку ви щойно бачили.

- Коли існує висока мінливість значень, але кількість елементів у списку постійна, ми можемо легко використовувати prepared оператори.

Труднощі виникають, коли значення сильно різняться, і кількість елементів у списку також сильно змінюється. Навіть якщо ви використовуєте prepared оператори та ретельно використовуєте маркери для кожного значення в списку, два оператори, які відрізняються лише кількістю елементів у списку, є різними операторами, які потрібно аналізувати окремо. У prepared операторів немає поняття «змінного списку аргументів», який допускають деякі мови.

Списки не були б великою проблемою, якби все, що у вас було, це одна умова в реченні where з критерієм in, який бере

свої значення зі випадного списку у формі HTML; навіть якщо ви можете вибрати кілька елементів, які ви можете мати у своєму списку, закгальна їх кількість обмежена, і неважко мати, скажімо, 10 різних операторів у кеші: один для списку з одним значенням, один для список із двома значеннями і так далі до списку з 10 значень.

Але тепер припустімо, що цей список є лише одним із п’яти різних критеріїв, які можна динамічно додавати для створення запиту. Ви повинні пам’ятати, що наявність п’яти можливих критеріїв означає, що ми можемо мати 32 (2↑5) комбінації, а мати 32 різні prepared оператори є цілком розумним. Але якщо одним із цих п’яти критеріїв є список, який може приймати будь-яку кількість значень від 1 до 10, цей список сам по собі еквівалентний 10 критеріям, а наші п’ять критеріїв фактично еквівалентні 14. Таким чином, 2↑14 = 16 284 , і ми вже не в такому порядку величин, як раніше. І, звісно, чим більша максимально можлива кількість елементів у списку, тим гірша кількість комбінацій. Можливості вибухають, а користь від використання prepared операторів зменшуєтьс.

Маючи це на увазі, існує три способи обробки списків у prepared операторах, це ми обговоримо у наступних трьох підрозділах.

Передача списку у вигляді однієї змінної

Використовуючи деякі засоби SQL, можна розділити рядок і зробити його схожим на таблицю, з якої можна вибирати або з’єднувати з іншими таблицями. Для подальших пояснень використаємо простий приклад у MySQL, який можна легко адаптувати до іншого діалекту SQL. Припустимо, що наша програма викликається формою HTML, у якій є список, з якого

користувач може вибрати кілька значень. Відомо лише те, що можна повернути не більше 10 значень.

Перша операція полягає в тому, щоб об’єднати всі вибрані значення в рядок за допомогою роздільника (у цьому випадку коми), а також додати до списку на початку і в кінці той самий роздільник, як у цьому прикладі:

```
',Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,'
```

Цей рядок буде параметром. Далі помножимо цей рядок стільки разів, скільки максимально очікується елементів, використовуючи декартове з’єднання наступним чином:

```
select pivot.n, list.val
from (select 1 as n
      union all
      select 2 as n
      union all
      select 3 as n
      union all
      select 4 as n
      union all
      select 5 as n
      union all
      select 6 as n
      union all
      select 7 as n
      union all
      select 8 as n
```

```
    union all
    select 9 as n
    union all
    select 10 as n) pivot,
  (select ? val) as list;
```

Є кілька способів отримати щось схоже на pivot. Наприклад, можна було б використати спеціальну таблицю для більшої кількості рядків. Якщо замінити рядок на маркер місця заповнювача і запустити запит, то отримаємо такий результат:

```
mysql> \. list0.sql
```

n	val
1	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
2	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
3	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
4	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
5	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
6	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
7	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
8	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
9	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,
10	Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey,

10 rows in set (0.00 sec)

Але в цьому списку тільки сім елементів. Тому потрібно обмежити кількість рядків кількістю елементів. Найпростіший спосіб підрахувати елементи — це обчислити довжину списку, видалити всі роздільники, обчислити нову довжину та отримати кількість роздільників із різниці.

```
where pivot.n < length(list.val) - length(replace(list.val, ',', ''));
```

Тепер треба, щоб перший елемент був у першому рядку, другий елемент у другому рядку і так далі. Якщо переписати запит, як показано тут:

```
select pivot.n, substring_index(list.val, ',', 1 + pivot.n)
from (select 1 as n
    union all
    select 2 as n
    union all
    select 3 as n
    union all
    select 4 as n
    union all
    select 5 as n
    union all
    select 6 as n
    union all
```

```
        select 7 as n
        union all
        select 8 as n
        union all
        select 9 as n
        union all
        select 10 as n) pivot,
      (select ? val) as list
  where pivot.n < length(list.val) - length(replace(list.val, ',', ''));
```

отримаємо наступний результат.

```
mysql> \. list2.sql
+---+-----------------------------------------------+
| n | substring_index(list.val, ',', 1 + pivot.n)   |
+---+-----------------------------------------------+
| 1 |  ,Doc                                         |
| 2 |  ,Doc,Grumpy                                  |
| 3 |  ,Doc,Grumpy,Happy                            |
| 4 |  ,Doc,Grumpy,Happy,Sneezy                     |
| 5 |  ,Doc,Grumpy,Happy,Sneezy,Bashful             |
| 6 |  ,Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy      |
| 7 |  ,Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey |
+---+-----------------------------------------------+
7 rows in set (0.01 sec)
```

Тепер досить близько до кінцевого результату, який можна отримати, застосувавши substring_index() вдруге. Потрібно, щоб substring_index() тепер повертав останній елемент у кожному рядку, що і робиться заміною першого рядка у запиті на таке:

```
select pivot.n,
        substring_index(substring_index(list.val, ',', 1 + pivot.n), ',', -1)
```

В результаті отримуємо:

```
mysql> \. list3.sql
+---+-----------------------------------------------------------------------+
| n | substring_index(substring_index(list.val, ',', 1 + pivot.n), ',', -1) |
+---+-----------------------------------------------------------------------+
| 1 |  Doc                                                                  |
| 2 |  Grumpy                                                               |
| 3 |  Happy                                                                |
| 4  | Sneezy                                                                |
| 5  | Bashful                                                               |
| 6  | Sleepy                                                                |
| 7  | Dopey                                                                 |
+---+-----------------------------------------------------------------------+
7 rows in set (0.00 sec)

mysql>
```

Як ви могли помітити, незважаючи на те, що запит є складнішим, ніж середній запит підручника SQL, він працює, як показано тут, із будь-яким списком, який містить до 10

елементів, і його можна легко адаптувати для обробки більших списків. Текст оператора залишається ідентичним і не підлягає повторному аналізу, навіть якщо кількість елементів у списку змінюється. Передача значень списку як одного рядка є дуже гарним і ефективним рішенням для невеликих списків. Його обмеженням є довжина рядка символів: ви можете натиснути на обмеження або в мові обертання на кожен, або в SQL (Oracle має найкоротший ліміт, 4000 символів); якщо ваш список містить кілька сотень елементів, тоді інше рішення може бути більш доцільним.

Пакетна обробка списків

Одне з рішень полягає в тому, щоб працювати партіями, намагаючись зберегти списки з фіксованою кількістю елементів. Те, що дозволяє нам використовувати такий метод, полягає в тому, що in() ігнорує дублікати в списку, чи то явний список, чи результат підзапиту. Припустимо, ми отримуємо наші параметри в масиві, який, як і в попередньому прикладі, може містити до 10 значень. М'яке кодування оператора:

```
my_condition = "where name in"
loop for i in 1 to count(myarray)
    if i > 1 then my_condition = my_condition + "("
                else my_condition = my_condition + ","
    my_condition = my_condition + "?"
end loop
```

```
my_condition = my_condition + ")"
```

Потім в новому циклі зв’яжемо кожне значення з його маркером місця і отримаємо стільки різних операторів, скільки елементів може бути у списку. Замість цього можна згенерувати оператор, який зможе обробляти до 10 елементів у списку:

```
my_condition = "where name in (?,?,?,?,?,?,?,?,?,?)"
```

Далі можна підготувати оператор, виконавши відповідний виклик до БД, і послідовно прив’язати 10 параметрів, повторивши деякі з фактичних параметрів, щоб заповнит 10 елементів:

```
loop for i in 1 to 10
    j = 1 + modulo(i, count(array))*
    bind(stmt_handler, i, array(j))

end loop
```

При потребі ми можемо мати кілька розмірів списків, з одним оператором для розміщення, скажімо, списків із 1 до 20 елементів, іншим оператором для списків із 21 до 40 елементів і т.д. Таким чином, коли ми отримуємо списки, що містять будь-яку кількість елементів до 100, ми впевнені, що збережемо в кеші щонайбільше 5 різних операторів замість 100.

Використання тимчасової таблиці

Для дуже великих списків найпростішим рішенням є, ймовірно, використання тимчасової таблиці та заміна явного списку підзапитом, який вибирає з цієї тимчасової таблиці, знову ж таки, текст запиту залишатиметься незмінним, незалежно від кількості рядків. у тимчасовій таблиці. Недоліком цього методу є вартість вставки: це додає більше взаємодії з базою даних. Однак справжнє питання стосується походження даних, які використовуються для заповнення списку. Для величезних списків нерідко елементи надходять із запиту. Якщо це так, insert ... select ... у тимчасову таблицю буде повністю виконано на сервері та заощадить отримання даних у прикладній програмі. На цьому етапі ви можете задатися питанням, чи дійсно потрібна тимчасова таблиця, і чи простий підзапит або об'єднання не зробить цей трюк набагато ефективнішим. Якщо дані надходять із зовнішнього файлу і якщо цей файл можна надіслати на сервер, на якому розміщена база даних, ви можете використовувати різні утиліти для майже безболісного завантаження тимчасової таблиці. (Зовнішні таблиці Oracle, інструкція MySQL щодо завантаження даних у файлі та служби інтеграції SQL Server надають розробникам інструменти, які можуть усунути багато програмування.)

Bulk Operations

Крім індексування, статистики та швидкості синтаксичного аналізу, які є загальними моментами, що впливають на одну конкретну програму або всі програми, які отримують доступ до бази даних, існують інші типи процесів, які слід перевіряти, якщо продуктивність не така висока, як очікувалося. Наприклад, одним із факторів, який може значно уповільнити продуктивність під час отримання (або вставки) великої кількості рядків, є кількість двосторонніх передачі даних, особливо коли існує глобальна мережа (WAN) між сервером застосунків і БД.

Тут обговорюється тема, яка залежить не лише від СУБД, а й від мови, яка використовується для доступу до цієї СУБД; тому обговоримо це більш детально, щоб вказати на можливі проблеми та навести кілька прикладів. Принцип дуже простий: припустимо, вам потрібно перемістити купу піску. Робота рядок за рядком подібна до переміщення купи піску чайною ложкою. Можливо, використання лопати чи тачки було б ефективнішим.

Реалізації відрізняються. Деякі залишають більше на волю уяви, ніж інші, але в усіх випадках, коли ви отримуєте дані із сервера в циклі курсору, ви надсилаєте виклики, які, по суті, говорять «надішліть мені більше даних». На дуже низькому рівні ви надсилаєте в мережу пакет, який містить цю команду. Натомість СУБД надсилає інший пакет, який або містить потрібні вам дані, або просто повідомляє «Я готовий». Кожного

разу ви маєте зворотний зв’язок і, як наслідок, певну затримку мережі. Такі самі типи обміну відбуваються, коли ви вставляєте рядок за рядком даних. Пакети, якими обмінюються, мають фіксований розмір, і надсилання повного пакету не є ані дорожчим, ані повільнішим, ніж надсилання переважно порожнього пакета; обмін повнішими пакетами означатиме менше обмінів для того самого обсягу даних. Коли ви вставляєте або повертаєте величезні обсяги даних, наприклад, щоб створити файл, який потрібно надіслати в інше місце для подальшої обробки іншою системою, немає сенсу надсилати рядок за рядком на сервер або отримувати рядок за рядком із сервер, навіть якщо процедурна частина вашої програми фактично обробляє один рядок за раз. Деякі продукти та програмні середовища дозволяють пакетно виконувати операції; це зазвичай стосується, наприклад, T-SQL, JDBC та SQLJ, які дозволяють групувати кілька операторів вставки перед надсиланням їх на сервер, або об’єкта SqlPipe у .NET Framework. Інші дозволяють вставляти з масивів або отримувати в них, як у випадку з Oracle і PL/SQL або інтерфейсом OCI C/C++. MySQL використовує дещо інший підхід до потокової передачі; він надсилає безперервний потік даних без явного запиту на це. Наприклад, якщо ви використовуєте бібліотеки C або PHP, ви отримаєте весь набір результатів одразу, що добре для набору результатів малого чи середнього розміру, але може призвести до нестачі пам’яті, коли вам справді потрібно повернути багато даних , у цьому випадку необхідно перейти в інший режим

У деяких випадках пакетування неявно виконується клієнтською стороною вашої програми. Наприклад, ви можете закодувати це так:

```
while fetch_one_row_from_database( )
        do something
```

і за вашою спиною сторона клієнта зробить наступне:

```
while fetch_one_row_from_array( )
        do something

fetch_one_row_from_array {
        increase index;
        if index > array_size
                fetch_array_size_rows_at_once_from_database
                set index to point to first array entry
        if buffer[index] is empty
                return done
        else
                return buffer(index)
```

Зазвичай це відбувається в програмі JDBC, у функціях інтерфейсу бази даних PHP або в деяких клієнтських бібліотеках C/C++. Мало хто з розробників пам’ятає, що розмір масиву іноді

знаходиться під їхнім контролем і що значення за замовчуванням не завжди найкраще підходить для того, що вони мають робити.

На рис 4 показано, як на продуктивність впливає встановлення «розміру вибірки» у випадку програми JDBC, яка створює дамп вмісту таблиці транзакцій. Не слід намагайтися порівнювати різні продукти один з одним, оскільки всі тести не проводилися на одній машині, і нормалізація кількісті рядків, що повертаються за одиницю часу, здійснювалась у «індекс пропускної здатності». MySQL і SQL Server (тобто їх драйвери JDBC) явно ігнорують налаштування, але Oracle надзвичайно чутливий до параметра. Збільшення розміру вибірки від значення за замовчуванням 10 до 100 подвоює пропускну здатність у цьому прикладі з дуже невеликими зусиллями (і програма JDBC, і база даних працювали на одній машині). Коли зазначалося, що MySQL ігнорує налаштування, це було не зовсім правдою: вдалося підвищити пропускну здатність MySQL приблизно на 15%, створивши оператор типу *forward only* і *read only*, і встановивши для розміру вибірки значення Integer.MIN_VALUE, що викликало перехід до швидшого (в цьому випадку) потокового режиму (який, як виявилося, також підтримується протоколом TDS SQL Server).

Вплив setFetchSize() з JDBC Driver

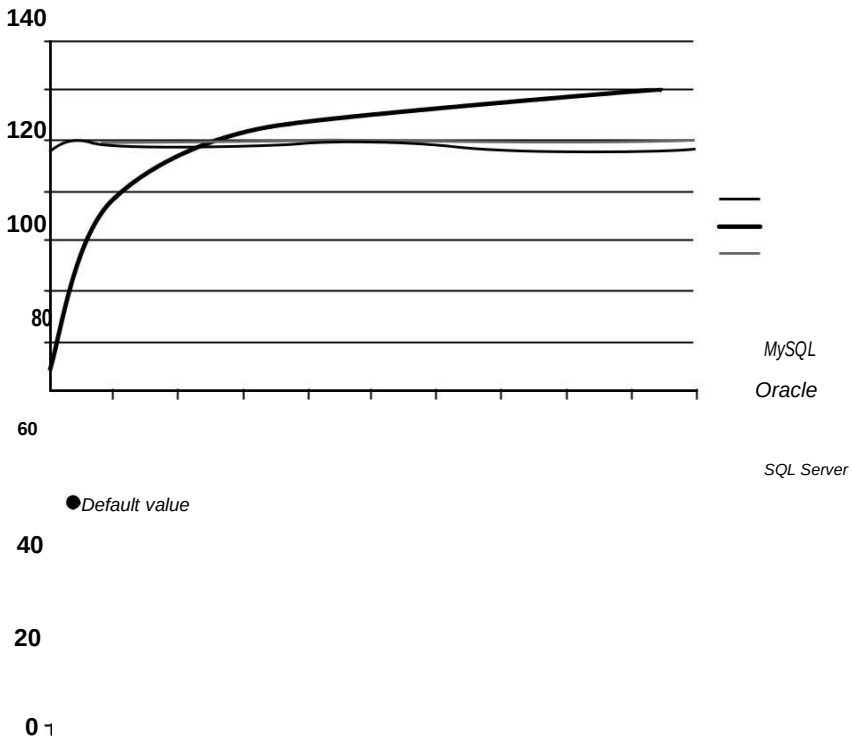

Рис. 4 . Зміна розміру вибірки у програмі JDBC

Усі згадані випадки мають досить консервативний підхід до групових операцій, здебільшого буферизуючи дані для оптимізації передачі. Ви також можете зіткнутися з масовими операціями, які виходять за межі буферизації та сміливо обходять звичайні операції SQL, що також може бути дуже цікавим для масивних процесів: наприклад, операції масового копіювання за допомогою Усі випадки, які я згадав досі, мають досить консервативний підхід до групових операцій, здебільшого буферизуючи дані для оптимізації передачі. Ви також можете зіткнутися з масовими операціями, які виходять за межі буферизації та сміливо обходять звичайні операції SQL, що також може бути дуже цікавим для масивних процесів: наприклад, операції масового копіювання за допомогою SQL Server's SQL Native Client або функції *direct path-loading* Oracle's C call interface. Але в цьому випадку ми дуже далекі від швидких, простих змін

Управління транзакціями

Ще один важливий момент, який слід перевірити, це управління транзакціями. Транзакція - це набір операторів break-or-pass, які відкриваються або неявно першим оператором, який змінює базу даних, або явно, і завершуються або фіксацією, яка робить зміну постійною на випадок збою сервера або відкату, який скасовує всі зміни в базі даних з початку транзакції (або додаткову проміжну точку збереження). Завершення транзакції знімає блокування, отримані сеансом для таблиці, сторінки або рядка, які були змінені, залежно від рівня блокування. На практиці, якщо

сервер аварійно завершує роботу в середині транзакції, будь-які зміни втрачаються, оскільки вони не обов’язково були записані на диск (якщо вони були записані, їх буде відкочено під час запуску бази даних).

Фіксація потребує часу, тому що єдиний спосіб гарантувати, що зміни будуть постійними, навіть якщо база даних виходить з ладу, — це записати всі оновлення в файл в енергонезалежну пам’ять з підтвердженням успішного завершення процесу запису даних. Часті зміни приведуть до того, що значна частина вашого часу займає їх очікування.

У середовищах онлайн-обробки транзакцій (OLTP) критично важлива часта фіксація, оскільки якщо блокування утримуються довше, ніж необхідно, одночасні транзакції серіалізуються та накопичуються, очікуючи завершення попередньої транзакції та розблокування ресурсу. Це дещо відрізняється під час нічних пакетних оновлень, коли паралелізму немає, і один процес вставляє, очищає або оновлює дані у великих масштабах. Якщо ви не впевнені, як часто ваша пакетна програма завантаження даних або оновлення виконується під час виконання, це обов’язково варто перевірити. За замовчуванням у деяких мовах (JDBC!) використовується режим «автоматичної фіксації», тобто фіксація здійснюється після кожної зміни в БД, що є надзвичайно болючим для всіх продуктів БД.