АНОТАЦІЯ

Тема магістерської роботи «Розробка підсистеми дослідження аналізу алгоритмів оптимізації виконання SQL-запитів».

Актуальність магістерської роботи полягає в необхідності аналізу великих обсягів даних в реальному часі, обсяги яких значно перевищують можливості сучасних інформаційних технологій.

Об'єкт дослідження – процеси аналізу алгоритмів оптимізації виконання SQL-запитів у базах даних.

Мета роботи – розробка підсистеми, яка допоможе дослідити та оптимізувати виконання SQL-запитів у базах даних за заданим критерієм.

Для реалізації поставленої мети були вирішені наступні питання: проведено дослідження механізмів доступу до таблиць баз даних; виконано аналіз алгоритмів реалізації операції вибірки та з’єднання таблиць баз даних; досліджені способи аналізу плану виконання запитів; виконано проектування та програмна реалізація підсистеми аналізу алгоритмів оптимізації виконання SQL-запитів. Практична цінність магістерської роботи полягає в тому, що розроблену підсистему зручно використовувати для дослідження ефективності використання SQL-запитів в базах даних.

Ключові слова: інформаційна підсистема, SQL-запит, алгоритмів оптимізації, проектування.

Магістерська робота містить в собі 60 сторінок, 34 рисунка, 14 посилань та 15 листів додатків.

ANNOTATION

The topic of master work «Development subsystem research analysis algorithms optimize performance of SQL-queries».

The urgency of the master's thesis is the need to analyze large amounts of data in real time, the amount of which far exceed the capabilities of modern information technology.

Object of research – process analysis of algorithms optimize performance of SQL-queries to databases.

Purpose – to develop subsystems that helps explore and optimize performance of SQL-queries in databases for a given criterion.

To achieve this goal have been resolved following questions: a study of the mechanisms of access to database tables; the analysis of algorithms implementa-

tion and operation of the sample connection database tables; study ways of analyzing the query plan; completed the design and implementation of a software subsystem analysis algorithms optimize the performance of SQL-queries. Practical value of master's thesis is that the developed subsystem is useful to study efficiency of SQL-queries in databases.

Keywords: information subsystem, SQL-query optimization algorithms, design.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 9
1 ДОСЛІДЖЕННЯ МЕХАНІЗМІВ ОПТИМІЗАЦІЇ SQL-ЗАПИТІВ 10
1.1 Механізми доступу для таблиць баз даних 10
1.2 Алгоритми реалізації операції вибірки 12
1.3 Аналіз алгоритмів з'єднання таблиць 13
1.4 Дослідження типів вбудованих оптимізаторів СУБД 16
1.4.1 Оптимизатор за синтаксисом 17
1.4.2 Оптимізатор за вартістю 18
1.5 Механизм статистики СУБД 18
1.5.1 Вплив на оптимізатор за вартістю 20
1.6 Визначення та завдання режиму оптимізатора 21
2 СПОСОБИ АНАЛІЗУ ПЛАНУ ВИКОНАННЯ ЗАПИТІВ 23
2.1 Визначення неефективної статистики 24
2.2 Визначення проблеми з індексами 25
2.3 Визначення причин неефективно складеного запиту 28
3 ВИБІР ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ ПІДСИСТЕМИ 30
3.1 Система управління базою даних Oracle 30
3.2 Мова програмування Java 34
3.3 Бібліотека Swing 35
3.4 Автоматизация процесса сборки 36
4 ПРОЕКТУВАННЯ ПІДСИСТЕМИ АНАЛІЗУ АЛГОРИТМІВ ОПТИМІЗАЦІЇ 38
4.1 Проектування підсистеми за допомогою методології функціонального моделювання SADT 38
4.2 Проектування підсистеми за допомогою послідовного виконання процесів Workflow Diagramming 43
4.3 Проектування підсистеми за допомогою діаграми потоків даних DFD . 45
5 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПІДСИСТЕМИ 47
5.1 Розробка інтерфейсу підсистеми 47
5.2 Тестування алгоритмів оптимізації плану запитів 49
5.2.1 Дослідження плану виконання запитів шляхом установки вбудованих режимів оптимізації 49
5.2.2 Оптимізація та дослідження плану виконання запитів шляхом виправлення неефективної структури запиту 54

5.2.3 Примусове використання вбудованих підказок (хинтів).................. 56
ВИСНОВКИ.. 60
ПЕРЕЛІК ПОСИЛАНЬ 61
ДОДАТОК А Основні коди програмних модулів……………………………..61

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

CSS – Cascading Style Sheets – каскадні таблиці стилів

HTML – HyperText Markup Language – мова гіпертекстової розмітки

JPA – Java Persistence API – стандартизований інтерфейс для Java ORM фреймворків

JSF – JavaServer Faces –технологія для веб-застосувань, що написані на Java

ORM – Object-relational mapping – технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних»

JSON – JavaScript Object Notation – текстовий формат обміну даними

AJAX – Asynchronous Javascript and XML – підхід до побудови інтерактивних користувацьких інтерфейсів веб-додатків, що полягає в «фоновому» обміні даними браузера з веб-сервером.

JSP – JavaServer Pages – технологія, що дозволяє створювати вміст, який має як статичні, так і динамічні компоненти.

EJB – Enterprise JavaBeans – специфікація технології написання та підтримки серверних компонентів, що містять бізнес-логіку.

POJO – Plain Old Java Object – простий Java-об'єкт, що не успадкований від якогось специфічного об'єкта і не реалізує жодних службових інтерфейсів понад ті, які потрібні для бізнес-моделі.

Apache – вільний веб-сервер

MySQL – вільна система управління базами даних (СКБД).

Java – об'єктно-орієнтована мова програмування.

Hibernate – засіб відображення між об'єктами та реляційними структурами для платформи Java.

Хостинг – послуга з надання простору для розміщення сайтів в Інтернеті.

ВСТУП

Проблема ефективного аналізу великих обсягів даних в реальному часі залишається актуальною протягом десятиліть. Проблема полягає в тому, що обсяги даних значно перевищують можливості їх обробки. З одного боку, на додаток до швидкого зростання обсягів накопиченої щодо статистичної інформації, зокрема, в Інтернеті, стрімко зростають обсяги доступної динамічної інформації, наприклад, в соціальних мережах [1].

З іншого боку, можливості обробки даних розвиваються повільніше: це відноситься як до технічних можливостей обладнання і алгоритму аналізу даних, так і до засобів специфікації і програмної реалізації необхідних обчислень. Системи, що інтегрують у собі різні підходи до аналізу даних, як правило, використовують проміжні алгебраїчні мови, на додаток до базової виразності традиційних декларативних мов запитів: фільтрації, теоретико-множинних операцій і операцій з'єднання.

Швидкість генерації та накопичення даних, наприклад в соціальних мережах і різноманітними сенсорами, веде до стрімкого збільшення обсягів аналізуємих даних. Це призводить до необхідності вирішення питань, пов'язаних зі швидкістю аналізу, так як багато в чому саме зростання обсягів даних призводить до збільшення часу їх обробки. Необхідність збільшення швидкості аналізу даних, що диктується реальністю предметних областей, має на увазі не тільки збільшення обчислювальних потужностей, але і опрацювання на інших рівнях [2].

Можливий підхід до вирішення цієї проблеми – аналіз великих обсягів даних заснований на використанні декларативних мов запитів(SQL). Подібні мови використовуються в традиційних системах баз даних протягом десятиліть. вони забезпечують не тільки можливості високорівневої специфікації необхідних обчислень, але і високу ефективність їх виконання, тому що відкривають широкі можливості для автоматичного вибору найбільш ефективних алгоритмів масової обробки даних.

1 ДОСЛІДЖЕННЯ МЕХАНІЗМІВ ОПТИМІЗАЦІЇ SQL-ЗАПИТІВ

Оптимізація SQL-запитів – це функція СУБД, що здійснює пошук оптимального плану виконання запитів з усіх можливих для заданого запиту, процес зміни запиту і / або структури БД з метою зменшення використання обчислювальних ресурсів при виконанні запиту. Один і той же результат може бути отриманий СУБД різними способами (планами виконання запитів), які можуть істотно відрізнятися як за витратами ресурсів, так і за часом виконання. Завдання оптимізації полягає в знаходженні оптимального способу.

В процесі виконання запиту після того як, парсер виконає синтаксичний аналіз запиту для перевірки правильності конструкцій запиту та семантичний аналіз для перевірки правильності назв об’єктів,типів об’єктів використаних в запиті, СУБД будує план виконання запиту (execution plan) [1].

План виконання запиту – послідовність операцій, необхідних для отримання результату SQL-запиту в реляційної СУБД, який містить логічний план виконання та фізичний план виконання.

Логічний план виконання – дерево реляційної алгебри, в вузлах якого розташовані операції реляційної алгебри(проекція, з'єднання, вибірка);

Фізичний план виконання – розвиток дерева реляційної алгебри, в вузлах якого розташовані алгоритми реалізації операцій реляційної алгебри.

План виконання включає шлях доступу (access path) для кожної таблиці із запиту і пов'язує таблиці (join order) за відповідними методам зв'язку (join method).

1.1 Механізми доступу для таблиць баз даних

Алгоритм табличного сканування (Table Scan) – доступ до даних через послідовну вибірку всіх рядків таблиці. Кількість операцій доступу залежить від кількості рядків таблиці.

Алгоритм індексного сканування (Index Scan) – доступ до даних через послідовний доступ до рядків індексу [3].

Індекс – об'єкт бази даних, що зберігає записи таблиці у вигляді структури <ROWID, колонка>, де ROWID – фізична адреса рядка таблиці в блоках БД, що визначає: адреса файлу, адреса сторінки в файлі і номер позиції в сторінці, колонка – відсортоване значення колонки. Індекс створюється з метою підвищення продуктивності пошуку даних. Таблиці в базі даних можуть мати велику кількість рядків, які зберігаються в довільному порядку, і їх по-

шук по заданому критерію шляхом послідовного перегляду таблиці рядок за рядком може займати багато часу. Індекс формується із значень одного або декількох стовпців таблиці і покажчиків на відповідні рядки таблиці і, таким чином, дозволяє шукати рядки, що задовольняють критерію пошуку. Прискорення роботи з використанням індексів досягається в першу чергу за рахунок того, що індекс має структуру, оптимізовану під пошук – наприклад, збалансованого дерева. Типи індексів:

– UNIQUE – бінарне дерево з відсортованими значеннями колонок, яке гарантуватиме унікальність значень, що створюється автоматично для обмежень типу PRIMARY KEY або UNIQUE;
– функціональний – як параметр використовує будь-яку функцію перетворення даних;
– BITMAP – ефективний для колонок з невеликою кількістю унікальних значень (не більше 10) і формує бітову матрицю, колонками якої є унікальні значення, рядками всі рядки таблиці,а осередки дорівнюють 1 або 0.

Вибір індексу. Для вибору індексу для кожної таблиці насамперед знаходяться всі потенційні індекси, які можуть бути застосовані в досліджуваному запиті. Оскільки ключі в індексі впорядковані, то ефективна вибірка з нього може виконуватися тільки в лексикографічному порядку. У зв'язку з цим, вибір індексу ґрунтується на наявності обмежень для колонок, що входять в індекс, починаючи з першої.

Для кожної колонки, що входить до індексу, починаючи з першої, шукаються обмеження з усього набору обмежень для даної таблиці, включаючи умови з'єднань. Якщо для першої колонки індексу не може бути знайдено жодного обмеження, то індекс не використовується (в іншому випадку довелося б сканувати індекс цілком). Якщо для чергової колонки обмежень не може бути знайдено, то пошук завершується і індекс приймається.

При оцінці планів виконання досліджуються альтернативні набори індексів, які можуть бути використані для вибірки. У разі вкладених циклів найбільш зовнішній цикл не може використовувати жодного індексу, який обмежується хоча б однією умовою сполуки, оскільки при виконанні цього циклу жодна з умов з'єднання повністю не визначено. Внутрішній цикл не може використовувати жодного індексу з обмеженнями, не сумісними з умовами з'єднання.

Решта індексів ранжується за кількістю видобутих рядків і довжині ключа.

Очевидно, число видобутих рядків залежить від обмежень. Чим менше число видобутих рядків і коротше довжина ключа, тим вище ранг. Індекс з найвищим рангом використовується для вибірки.

Сканування індексу цілком.

Для виконання деяких запитів з агрегацією індекс може скануватися цілком. Зокрема:

– для пошуку глобальних максимальних і мінімальних значень використовуватися індекс по відповідній колонці (колонок) без обмежень;
– для пошуку числа рядків в таблиці використовується індекс по первинному ключу, якщо такий є. Це пов'язано з тим, що СУБД не зберігає і не може зберігати число рядків в таблиці, а сканування індексу по первинному ключу найменш ресурсоємних.

Якщо запитаний порядок вибірки збігається з порядком одного або більше індексів, то виконується оцінка можливості сканування одного з таких індексів цілком.

Якщо індекс містить всі колонки, необхідні для отримання результату, то сканування таблиці не відбувається. Якщо оптимізатор використовує цей фактор, то можна прискорити вибірку з таблиці для спеціалізованих запитів шляхом включення в індекс надлишкових колонок, які будуть вилучені безпосередньо при пошуку по ключу [4].

1.2 Алгоритми реалізації операції вибірки

Table Access Full – повне сканування таблиці (також відомий як послідовне сканування) є сканування бази даних, де кожен рядок таблиці при скануванні зчитується в послідовному порядку і стовпці, з якими стикаються перевіряються на дійсність умови. Таке сканування таблиць, як правило, є найповільнішим методом сканування таблиці через велику кількість необхідних I/O операцій читання з диска. Послідовне сканування відбувається зазвичай, коли стовпець або група стовпців таблиці (таблиця може бути на диску або може бути проміжним таблиця створена об'єднанням двох або більше таблиць), які необхідні для сканування не містять індекс [2]:

– table Access By User RowID – вибірка рядків з таблиці, якщо в предикату запиту зазначена умова з ROWID;
– index Full Scan – вибірка рядків на основі перегляду індексу за умови OT NULL для стовпця;

- table Access By Index ROWID – вибірка рядків з таблиці, яка використовує результати будь-якого алгоритму;
- index Scan – через зв'язування рядків індексу з рядками таблиці за значенням ROWID;
- index Fast Full Scan – вибірка рядків на основі перегляду індексу за умови отримання даних тільки з стовпців індексу без необхідності зв'язку з основною таблицею, тому алгоритм швидше ніж Index Full Scan;
- index Unique Scan – вибірка рядків на основі перегляду індексу за умови отримання унікальних значень стовпця, який пов'язаний з обмеженнями PRIMARY KEY або UNIQUE;
- index Range Scan – вибірка рядків на основі перегляду індексу, якщо в предикаті запиту є операції », «, = (при не унікальних значеннях стовпця), а також like 'pattern% _', при чому спеціальні символи повинні стояти після pattern;
- index Skip Scan – вибірка рядків на основі перегляду складеного індексу, якщо в предикаті запиту використовується колонка, розташована в індексі не на першому місці.

1.3 Аналіз алгоритмів з’єднання таблиць

Дослідимо алгоритм з'єднання таблиць «вкладені цикли». Цей алгоритм складається з наступних кроків:

Визначення провідної таблиці (зовнішня таблиця – таблиця на зовнішній стороні з'єднання) і відомої таблиці (внутрішня таблиця).

Для кожного рядка провідною таблиці вибираються всі рядки в відомою таблиці:

- операція складається з зовнішнього циклу (зовнішній контур) по кожному рядку провідною таблиці і внутрішнього циклу (внутрішній цикл) по кожному рядку відомою таблиці;
- в плані виконання зовнішній цикл (зовнішній контур) показується перед внутрішнім (внутрішній контур).

Детальний опис алгоритму. Алгоритм складається з довільного числа вкладених ітерацій пошуку даних в кожній з з'єднуються таблиць. Зовнішнім циклом виконується пошук рядків у провідній таблиці. Якщо частина або всі обмеження для провідної таблиці можуть бути використані для пошуку за індексом, то на кожній ітерації циклу в індексі шукаються розташування всіх

необхідних рядків і виконується прямий доступ до таблиці. В іншому випадку таблиця сканується цілком. Решта обмеження використовуються для фільтрації обраних рядків. Для кожної залишилася рядка викликається внутрішній цикл. Внутрішній цикл за умовами з'єднання і даними зовнішнього циклу шукає рядки в відомою таблиці. Якщо частина або всі обмеження для відомою таблиці разом з обмеженнями, отриманими від зовнішнього циклу, можуть бути використані для пошуку за індексом, то на кожній ітерації циклу в індексі шукаються розташування всіх необхідних рядків і виконується прямий доступ до відомою таблиці. В іншому випадку таблиця сканується цілком. Решта обмеження використовуються для фільтрації обраних рядків. На кожній ітерації найглибшого циклу вибрані з таблиць рядки конкатенуються, для отримання рядків підсумкового результату.

У загальному випадку цикли можуть вкладатися довільне число раз, в залежності від числа таблиць, що беруть участь в з'єднанні. Якщо в деякому циклі виконується пошук за індексом, і всіх колонок в індексі досить для отримання підсумкового результату, то прямий доступ до таблиці в цьому циклі не виконується [5].

Переваги: найбільш загальний і тому незамінний алгоритм з'єднання. За допомогою з'єднання вкладеними циклами можна реалізувати будь-яку умову з'єднання. (Решта алгоритмів мають обмеження по реалізованим з їх допомогою умов з'єднання. Наприклад, коли умова виражається нерівністю). Найшвидший алгоритм, якщо необхідно отримати тільки перший рядок результату (наприклад в SQL виразах типу EXISTS). При використанні пошуку за індексом алгоритм краще всіх масштабується. Тобто при збільшенні обсягу даних в з'єднуються таблицях час виконання запиту збільшується практично лінійно при тих же апаратних ресурсах.

Недоліки: найповільніший алгоритм. Всі інші алгоритми мають перевагу в швидкості (але тільки в суворо визначених обставинах).

Дослідимо алгоритм з'єднання таблиць «сортування зі злиттям».

Сортування зі злиттям успішно використовується, коли в умовах з’єднання двох таблиць присутні оператори порівняння: <, <=, > або > =.

Швидкість виконання методу сортування зі злиттям більше, ніж у методу вкладеного циклу для великих наборів даних.

Алгоритм з'єднання таблиць «сортування зі злиттям» кроки алгоритму:

– sort join: сортування вихідних таблиць по ключу з'єднання (join key);
– з'єднання злиттям: злиття (об'єднання) таблиць по значеннях відсортованих стовпців. У випадку, якщо джерело даних вже впорядкова-

но по стовпцю операція Sort join не проводиться для цього джерела даних.

Таким чином алгоритм отримує на вхід дві таблиці і умова з'єднання. Результатом його роботи є таблиця з результатами з'єднання. Вхідні таблиці повинні бути відсортовані за стовпцями, які беруть участь в умови з'єднання. З'єднання здійснюється за одне сканування (прохід по) кожної з вхідних таблиць. Тобто один і той ж рядок зчитується тільки один раз, що дає перевагу перед з'єднанням вкладеними циклами.

Переваги: з'єднання злиттям ефективніше, ніж алгоритм з'єднання вкладеними циклами, за умови, що списки спочатку були відсортовані. В принципі, накладні витрати на сортування можуть бути розподілені між кількома операціями з'єднання. З'єднання злиттям на відміну від з'єднання з використанням хешування може використовуватися при великих розмірах з'єднуємих таблиць. З'єднання злиттям може використовуватися для з'єднань з умовами відмінними від рівності, чого не дозволяє алгоритм з'єднання хешем.

Недоліки: головним недоліком алгоритму є необхідність попереднього сортування списків. Накладні витрати на сортування можуть бути неприйнятно високими. При реалізації в СУБД, з'єднання злиттям вимагає більше пам'яті і менш гнучке, ніж алгоритм з'єднання вкладеними циклами.

У зв'язку з цим на практиці рекомендують уникати цього виду з'єднання. У багатьох СУБД з'єднання злиттям за замовчуванням не використовується оптимізатором запитів і має бути включено явно.

Дослідимо алгоритм з'єднання таблиць за хеш-значенням – «хешування». Хешування використовується для з'єднання великих наборів даних.

Кроки алгоритму:

– для меншої з двох таблиць в пам'яті будується хеш таблиця ключа з'єднання (join key);
– сканується велика таблиця і порівнюється за ключем з хеш-таблицею для отримання результуючого набору рядків.

Таким чином алгоритм отримує на вхід дві таблиці і умова з'єднання. Результатом його роботи є таблиця з результатами з'єднання.

Переваги: з'єднання хешуванням істотно швидше з'єднання вкладеними циклами. При відносно невеликому розмірі меншому таблиці це найефективніший вид з'єднання.

Недоліки: умовою з'єднання може бути тільки рівність.

Велика потреба в пам'яті для побудови хеш-таблиці, що вкрай обмежує масштабованість алгоритму при збільшенні розмірів меншою таблиці.

Хеш-таблиця повинна бути побудована повністю, до того як перший результат буде записаний в результуючу таблицю, що робить цей вид з'єднання неприйнятним при необхідності отримати перший рядок результату якомога швидше.

У реальних системах використовуються більш витончені схеми хешування, в основному націлені на зменшення потреби в пам'яті для побудови хеш-таблиці. Наприклад, дані обох таблиць розбиваються на частини, а хеш-таблиця будується тільки для однієї з цих частин.

Дослідимо алгоритм з'єднання таблиць «напівз'єднання» та «антиз'єднання».

«Напівз'єднання» – операція з'єднання (з'єднання алгоритмом вкладені цикли або хеш-з'єднання), яка повертає рядок провідної (зовнішньої таблиці при знаходженні першого збігу з рядком відомої (внутрішня таблиця) таблиці за умовами запиту (предикат). Використовується в запитах з операторами: "exists", "in".

«Антиз'єднання» – аналогічна операція, яка в разі знаходження першого збігу виключає рядок провідної (зовнішньої) таблиці з результатів пошуку. Використовується запитах з операторами "not exists","not in"[4].

1.4 Дослідження типів вбудованих оптимізаторів СУБД

В Oracle Database на вибір пропонується два різних оптимізатора запитів – за синтаксисом (rule-based optimizer) і за вартістю (cost-based optimizer). Наприклад, навіть у разі, коли запит стосується лише однієї таблиці, у сервера Oracle є такі можливості:

- знайти ROWID запитаних рядків за допомогою індексу, а потім витягти ці рядки з таблиці;
- переглянути всю таблицю і знайти відповідні рядки – це називається повним скануванням таблиці.

Хоча зазвичай вибірка за індексом набагато швидше, процес отримання значень з індексу вимагає додаткових операцій введення / виводу. Оптимізація запиту може іноді звестися до аналізу того, чи є в запиті умови щодо значень стовпців, що зберігаються в індексі. Використання значень з індексу для вибірки потрібних рядків вимагає менше операцій введення / виводу і тому виявляється ефективніше, ніж вилучення всіх даних з таблиці з подальшим

застосуванням до них умов відбору. Ще один фактор, який враховується при визначенні оптимального плану виконання запиту, – наявність в запиті пропозиції ORDER BY, яке можна було б реалізувати автоматично за рахунок використання вже відсортованого індексу. Навпаки, якщо таблиця мала, оптимізатор може вирішити, що вигідніше просто рахувати з неї всі блоки, проігнорувавши індекс, так як оцінна вартість введення /виведення для індексу і таблиці буде вищою, ніж для однієї лише таблиці. Оптимізатор повинен приймати важливі рішення, навіть коли в запиті бере участь тільки одна таблиця. Якщо ж запит складніший, наприклад включає багато з'єднуються між собою таблиць або містить заплутаний критерій відбору і кілька рівнів сортування, то труднощі стоїть перед оптимізатором завдання зростає багаторазово [6].

1.4.1 Оптимизатор за синтаксисом

В оптимізаторі за синтаксисом для прийняття рішення застосовується набір визначених правил. У деяких ситуаціях оптимізація за синтаксисом забезпечує кращу продуктивність, ніж ранні версії оптимізатора Oracle за вартістю. Але у оптимізатора за синтаксисом є слабкі сторони, одна з яких – надто спрощений набір правил. У складних базах даних запити часто будуються по декількох таблицях, які мають по кілька індексів, із застосуванням складних умов і угруповань. Результатом такої складності стає велика кількість шляхів виконання, і занадто простий набір правил не дозволяє зробити найкращий вибір. Оптимізатор за синтаксисом призначає кожному потенційному шляху виконання деяку оцінку, а потім вибирає шлях з найкращою оцінкою. Інша слабкість оптимізатора за синтаксисом проявляється в ситуації, коли є шляхи з однаковими оцінками. В цьому випадку оптимізатор дозволяє конфлікт за допомогою синтаксису SQL-запиту. Вибір остаточного шляху визначається тим, в якому порядку таблиці згадуються в SQL-запиті. До яких наслідків призводить такий алгоритм вирішення конфліктів, можна зрозуміти, розглянувши простий випадок, коли маленька таблиця SMALLTAB з 10 рядків з'єднується з великою таблицею LARGETAB. Якби оптимізатор вирішив спочатку читати SMALLTAB, то сервера довелося б вважати 10 рядків, а потім знайти в LARGETAB відповідні їм рядки [4].

Але якщо оптимізатор вважатиме, що спочатку слід читати LARGETAB, то сервер прочитає 10 000 рядків, а потім буде 10000 разів переглядати SMALLTAB в пошуках відповідностей. Звичайно, рядки з

SMALLTAB, швидше за все, виявляться в кеші, що скоротить витрати на кожен перегляд, але різниця в продуктивності проте буде величезною. Такі казуси залежать від порядку згадки таблиць в запиті. В описаній ситуації будуть повернуті одні і ті ж результати при обох шляхах виконання, але ресурси, витрачені на їх отримання, розрізняються досить істотно.

1.4.2 Оптимізатор за вартістю

Як випливає з назви, його завдання – не тільки простий вибір правил з заданого набору. Він вибирає шлях виконання, що вимагає найменшої кількості логічних операцій введення / виводу. Такий підхід дозволяє уникнути потенційних проблем, розглянутих вище. Оптимізатор оцінює вартість плану виконання за допомогою статистики, відноситься до потрібних йому структурам даних.

В Oracle Database статистика збирається за замовчуванням і зберігається в автоматичному репозиторії навантаження (Automatic Workload Repository,AWR). Статистичні дані містять інформацію про доступ до сегментів бази, статистику часової моделі, статистику системи і сеансів, відомості про SQL-запити, що призвели до найбільшого навантаження, і статистику історії активних сеансів (Active Session History, ASH).

Оптимізатор за вартістю знаходить оптимальний план виконання, привласнюючи оцінку кожному потенційному плану. При цьому він виходить з власних внутрішніх правил і логіки, а також з статистики, що відбиває стан структур даних в базі, враховується статистика для таблиць, стовпців і індексів, що беруть участь в плані виконання запиту [2].

1.5 Механизм статистики СУБД

Для оцінки потенційного числа рядків, що витягується з таблиці, СУБД використовує статистику. Статистика має вигляд гістограм для кожної колонки таблиці, де по горизонталі розташовується шкала значень, а висотою стовпчика відзначається оцінка числа рядків у відсотках від загального числа рядків. Таким чином, якщо з таблиці витягуються рядки зі значенням колонки з обмеженням [V1, V2], то можна оцінити число рядків, що потрапляють в цей інтервал.

Алгоритм оцінки числа видобутих рядків наступний:

– визначити, в які інтервали гістограми потрапляє обмеження [V1, V2];
– знайти оцінки числа рядків Ri для кожного інтервалу і в процентах;
– якщо [V1, V2] потрапляє в певний інтервал [S1, S2] частково або повністю лежить в інтервалі, то – знайти перетин [V1, V2] і [S1, S2] та відкоригувати число значень в частковому інтервалі;
– підсумувати оцінки у відсотках для всіх інтервалів;
– перевести оцінку у відсотках до числа рядків.

Як правило, СУБД не знає і не може знати точне число рядків в таблиці (навіть для виконання запиту SELECT COUNT (*) FROM TABLE виконується сканування первинного індексу), оскільки в базі можуть зберігатися одночасно кілька образів однієї і тієї ж таблиці з різним числом рядків. Для оцінки числа рядків використовуються наступні дані:

– число сторінок в таблиці;
– довжина сторінки;
– середня довжина рядка в таблиці.

Статистика також може зберігатися наростаючим підсумком. У цьому випадку кожен інтервал містить сумарну оцінку всіх попередніх інтервалів плюс власну оцінку.

Для отримання оцінки числа рядків для обмеження [V1, V2] досить з оцінки інтервалу, в який потрапляє V2, відняти оцінку інтервалу, в який потрапляє V1. Збір статистики для побудови гістограм здійснюється або спеціальними командами СУБД, або фоновими процесами СУБД. При цьому, з огляду на те, що база може містити суттєвий обсяг даних, робиться вибірка меншого обсягу з усієї генеральної сукупності рядків [7].

Оцінка репрезентативності (достовірності) вибірки може здійснюватися, наприклад, за критерієм згоди Колмогорова. Якщо дані в таблиці істотно змінюються в короткий проміжок часу, то статистика перестає бути актуальною і оптимізатор приймає неправильні рішення про повне сканування таблиць.

Режим роботи бази даних повинен бути спланований таким чином, щоб підтримувати актуальну статистику, або не використовувати оптимізацію на основі статистики. Статистика індексів відображає не тільки глибину і ширину дерева індексу, а й ступінь унікальності зберігаються в ньому значенні – від цього залежить, наскільки просто буде відібрати рядки з допомогою даного індексу.

Точність роботи оптимізатора за вартістю залежить від точності використовуваних їм статистик, тому своєчасне оновлення статистики було необхідно завжди. Неактуальна статистика може викликати зниження продуктивності.

Активізувати механізм збору статистики можна вибірково. Наприклад, в Oracle Database можна включити автоматичний збір статистики для таблиці, який починається, якщо статистика застаріла (тобто з моменту останнього збору змінилося більше 10 відсотків рядків в таблиці) або відсутня зовсім [6].

Статистика допомагає оптимізаторові приймати набагато більше осмислені рішення щодо вибору оптимального плану виконання. Наприклад, іноді оптимізаторові доводиться обирати між двома індексами, в яких можна було б шукати значення. Оптимізатор по синтаксису міг би поставити обом індексам однакові оцінки і вибрати план виконання виходячи з порядку їх появи в умові WHERE. Але оптимізатор за вартістю знає, що в одному індексі 1000 записів, а в іншому 10 000. Він знає навіть, що в першому індексі тільки 20 унікальних значень, а в другому – 5000. Стало бути, селективність більшого індексу набагато вище, тому саме він отримає вищу оцінку і буде використаний для виконання запиту.

Іноді нема необхідності оновлювати статистику, оскільки розподіл даних в базі досягло стабільного стану або запити і так виконуються оптимально (або, принаймні, з прийнятною продуктивністю).

Oracle дозволяє випробувати новий набір статистик і подивитися, чи дадуть вони більш високу продуктивність, залишаючи можливість повернутися до старого набору; в такому випадку можна зберегти статистики в окремій таблиці,а потім зібрати заново.

Якщо після тестування програми з новими статистиками буде зрозуміло, що попередні результати були краще, то можна просто відновити збережені статистики.

1.5.1 Вплив на оптимізатор за вартістю

Вплинути на те, як оптимізатор за вартістю обирає план виконання, можна двома способами. По-перше, за допомогою параметра ініціалізації OPTI-MIZER_MODE. За замовчуванням він дорівнює ALL_ROWS, і це означає прагнення до максимальної пропускної здатності. Значення FIRST_ROWS змушує оптимізатор вибрати план виконання, швидше за всіх

повертає перші рядки запиту. При цьому можна задати кількість обраних рядків.

Завдання режиму трохи змінює оцінки, що виставляються оптимізатором, що в деяких випадках може призвести до вибору іншого плану виконання. Вплинути на рішення оптимізатора можна і за допомогою підказок (hints).

Підказка (або хинт) – це не більше ніж коментар, задається в певному форматі в SQL-команді. Є такі категорії підказок:

- підказки, що пропонують змінити мета оптимізації;
- підказки про повне сканування таблиці;
- підказки про скануванні індексу;
- підказки про скануванні діапазону індексу по зменшенню;
- підказки про швидке повному скануванні індексу;
- підказки про з'єднання, в тому числі про з'єднання за індексом, з'єднанні за допомогою вкладених циклів, хеш-з'єднанні, з'єднанні сортуванням і злиттям, декартовом та інших видах сполучення;
- інші підказки, включаючи підказки про шляхи доступу, про перетворення запиту і про паралельному виконанні.

Якщо підказка знаходиться не в тому місці SQL-команди, або ключове слово написано неправильно, або ім'я структури даних змінилося, так що підказка більше не належить до існуючої структури, то вона буде просто проігноровані як звичайний коментар. Так як підказка є частиною SQL-команди, коли вона перестає працювати, знайти і виправити помилку дуже і дуже нелегко [8].

Крім того, якщо підказка включена, щоб обійти помилку оптимізатора, яку згодом усунуто, то при обробці запиту виправлений (і, можливо, покращений) оптимізатор все одно не буде використовуватися. Однак своя ніша у підказок є, наприклад, у разі, коли розробник визначив новий тип даних, що передбачає спеціальний тип доступу.

Оптимізатор не здатний передбачити особливості визначаються користувачем типів, але підказка, можливо, допоможе йому вибрати правильний шлях доступу.

1.6 Визначення та завдання режиму оптимізатора

RULE – змушує використовувати оптимізатор по синтаксису. CHOOSE – залишає вибір оптимізатора на розсуд сервера Oracle.

При роботі в режимі CHOOSE, який раніше приймався за замовчуванням, Oracle вибирав оптимізатор за вартістю, якщо хоча б для однієї зі згаданих у запиті таблиць була статистика. Для таблиць без статистики оптимізатор сам робив статистичні оцінки. Оптимізатор за вартістю приймає рішення, володіючи більш повною інформацією про структури в базі даних. Хоча його логіка і не бездоганна, він все ж робить набагато більш точні оцінки і вдосконалюється в кожній новій версії. В оптимізаторі за вартістю до того ж враховуються всі поліпшення і нововведення, що вносяться до СУБД Oracle [4]. Наприклад, він розуміє, як впливають на план виконання секціоновані таблиці. Оптимізатор по синтаксису нічого про них не знав. Оптимізатор за вартістю правильно будує план виконання запитів до схеми типу «зірка»,широко застосовується в сховищах даних, тоді як оптимізатор по синтаксису ні допрацьований з урахуванням таких запитів і багатьох інших особливостей аналітичних додатків. Крім того, у оптимізатора є три переваги:

– він бачить структуру всієї бази даних. Багато баз даних Oracle підтримують різноманітні програми та користувачів, так що цілком може статися, що система використовує дані спільно з якимись іншими, тому загальна структура і склад даних знаходяться поза контролем;
– оптимізатор бачить динамічно змінюється картину бази і зберігаються в ній даних. Використовувана їм статистика може змінюватися при кожному автоматизованому зборі.

Крім змінюваної статистичної інформації внутрішні механізми оптимізатора також пристосовуються до змін в роботі бази даних.

Починаючи з версії Oracle 9i оптимізатор за вартістю бере до уваги швидкодія ЦП, а з версії Oracle Database 10g враховує і загальну статистику підсистеми вводу / виводу [6].

Якщо потрібно перешкодити оптимізатору обчислювати новий план при кожному отриманні SQL-запиту, можна створити збережену схему плану виконання (stored outline), в якій запам'ятовуються атрибути, використовувані оптимізатором при побудові плану. Маючи збережену схему, оптимізатор просто застосує після успішної реєстрації атрибути для породження плану виконання.

2 СПОСОБИ АНАЛІЗУ ПЛАНУ ВИКОНАННЯ ЗАПИТІВ

При аналізі план проглядається від низу до верху. У процесі перегляду в першу чергу звертається увага на рядки з великими Cost, CPU Cost. Якщо з плану видно про різкий скачок цих значень – з цих рядків і треба починати пошук причини ресурсоємності запиту. Після знаходження рядків з великим Cost і CPU Cost триває перегляд плану від низу до верху до наступного великого CPU Cost і т.д. При цьому, якщо CPU Cost в рядку близький до CPU Cost першого рядка (максимальне значення), то знайдений рядок є визначальним в ресурсоємності запиту і з ним в першу чергу треба шукати причину ресурсоємності запиту.

Крім пошуку великих Cost і CPU Cost в рядках плану слід переглядати перший стовпець Operation плану на предмет наявності в ньому HASH JOIN. З'єднання по HASH JOIN призводить до з'єднання таблиць в пам'яті і, здавалося б, більш ефективним, ніж вкладені з'єднання NESTED LOOPS. Разом з тим, HASH JOIN ефективно при наявності таблиць, коли хоча б одна з яких поміщаються в пам'ять БД або при наявності з'єднання таблиць з нізкоселективними індексами. Недоліком цього з'єднання є те, що при нестачі пам'яті для таблиці (таблиць) будуть задіяні диски, які суттєво загальмують роботу запиту.

У зв'язку з чим, при наявності високоселективних індексів, доцільно подивитися, а не поліпшить план виконання хинт USE_NL, що приводить до з'єднання з вкладеним циклам NESTED LOOPS. Якщо план буде краще, то залишити цей хинт. При цьому в хинті USE_NL в дужках обов'язково повинні перераховуватися всі аліаси таблиць, що входять у фразу FROM, в іншому випадку може виникнути дефектний план з'єднання [6].

Цей хинт може бути посилений хинтами ORDERED і INDEX. Слід звернути також увагу на MER-GE JOIN. При великому CPU Cost в рядку з MERGE JOIN варто перевірити дію хинта USE_NL для поліпшення ефективності запиту.

Особливу увагу в плані слід так само приділити рядкам в плані з операціями повного сканування таблиць і індексів в стовпець Operation: FULL – для таблиць і FULL SCAN, FAST FULL SCAN, SKIP SCAN – для індексів. Причинами повного сканування можуть бути проблеми з індексами: відсутність індексів, неефективність індексів, неправильне їх застосування.

При невеликій кількості рядків в таблиці повне сканування таблиці FULL може бути нормальним явищем і ефективніше використання індексів.

Наявність в стовпці Operation операції MERGE JOIN CARTESIAN каже, що між якимись таблицями немає повної зв'язки. Ця операція виникає при наявності у фразі From трьох і більше таблиць, коли відсутні зв'язки між якийсь із пар таблиць.

Рішенням проблеми може бути додавання якої бракує зв'язки, іноді допомагає використання хинта Ordered. Після аналізу плану виконання запиту здійснюється його оптимізація [4].

Оптимізація запиту передбачає видалення причин неефективності запиту, серед яких найбільш вагомими є:

– погана статистика таблиць і індексів, що беруть участь в запиті (найбільш важливий фактор, на який в першу чергу треба звернути увагу);
– проблеми з індексами: відсутність потрібних індексів, неефективно побудовані індекси, неефективно використовуються індекси, велике значення фактора кластеризації;
– проблеми з хинтами: відсутність хинтов або вони неефективні;
– неефективна структура запиту (запит побудований не коректно).

2.1 Визначення неефективної статистики

Перш ніж оптимізувати запит, доцільно подивитися статистику таблиць і індексів, що беруть участь в запиті. Часом досить оновити статистику, щоб запит став працювати ефективно. Можливі варіанти не ефективною статистики, що призводять до ресурсоємності запиту.

Застаріла статистика. Час останнього збору статистики визначається значенням поля Last_Analyzed для таблиць і індексів, яке знаходитися з Oracle таблиць ALL_TABLES (DBA_TABLES) і ALL_INDEXES (DBA_INDEXES) відповідно. Oracle щодня в певні години в робочі дні та в певні години у вихідні сам збирає статистику по таблицях. Але для цього DML операції з таблицею повинні привести до зміни не менше 10% рядків таблиці.

Але існують ситуації, коли протягом дня таблиця неодноразово і істотно змінює число рядків або таблиця настільки велика, що 10% змін настає через тривалий час. В цьому випадку доводиться оновлювати статистику, використовуючи процедури збору статистики всередині пакетів, При цьому, чим вище відсоток збору, тим краще, однак, при цьому зростає і може бути суттєвим час збору статистики.

Відсутність статистики хоча б в одній з таблиць, що входять у фразу From може бути визначальним фактором ресурсоємності запиту. Це може статися, наприклад, в разі створення нової таблиці і використання її до моменту, коли Oracle в певні години сам збере статистику по таких таблиць [4].

Погана статистика таблиць і індексів. Як показала практика, поганий можна вважати статистику, коли відсоток збору статистики по таблиці або індексу менше 0.1%. Тоді необхідно перезібрати статистику по таблиці або індексу з поганою статистикою. Або застосувати такий прийом як блокування збору статистики. Використовується при інтенсивній зміні числа рядків в таблиці протягом дня (численні видалення і вставки рядків).

Блокування статистики: execute dbms_stats.lock_table_stats ('ім'я схеми', 'ім'я таблиці'); розблокування:execute dbms_stats.unlock_table_stats ('ім'я схеми', 'ім'я таблиці').

Також слід зазначити, що при хорошому значенні статистики по таблиці може бути неблагополучна статистика по якомусь індексу таблиці, в силу чого доцільно відстежувати статистику не тільки таблиці, але і індексів таблиці.

2.2 Визначення проблеми з індексами

Проблеми з індексами в плані виконання проявляються при наявності в стовпці Options значень FULL, FULL SCAN, FAST FULL SCAN і SKIP SCAN в силу наступних причин [5].

Відсутність потрібного індексу. Потрібні дії – створити новий індекс. Індекс є, але він неефективно побудований. Причинами неефективності індексу можуть бути:

– мала селективність стовпчика, на якому побудований індекс, тобто в стовпці багато однакових значень, мало унікальних значень. Рішення в даній ситуації – прибрати індекс з таблиці або стовпець, на основі якого побудований індекс, ввести в складовою індекс;
– стовпець селективний, але він входить в складову індексу, в якому цей стовпець не є першим (провідним) в індексі. Рішення – зробити цей стовпець провідним або створити новий індекс, де стовпчик буде ведучим;

Побудовано ефективний індекс, але він працює не ефективно в силу наступних причин:

– індекс заблокований від використання. Блокують використання індексу такі операції на колонку, за яким використовується індекс: SUBSTR, NVL, DECODE, TO_CHAR, TRUNC, TRIM, || конкатенація, + цифра до цифрового полю і т.д. Рішення – модифікувати запит, звільнитися від блокуючих операцій або створити індекс по функції, яка блокує індекс;
– не зібрана або неактуальна статистика за індексом. Рішення – зібрати статистику за індексом запуском процедури, зазначені вище;
– хинт, блокуючий роботу індексу, наприклад NO_INDEX;
– неефективно налаштовані параметри ініціалізації бази даних БД (особливо відповідальні за ефективну роботу індексів, наприклад, optimizer_index_caching і optimizer_index_cost_adj);
– є сильні індекси, але вони змагаються між собою. Це відбувається тоді, коли в умови where є рядок, в якому стовпець однієї таблиці дорівнює стовпчику іншої таблиці. При цьому на обох стовпчиках побудовані сильні або унікальні індекси. Наприклад, в умови Where є рядок AND A.ISN = B.ISN. При цьому обидва стовпчика ISN різних таблиць мають унікальні індекси.

Однак, ефективно може працювати індекс тільки одного стовпчика (лівого або правого в рівність). Індекс іншого в кращому випадку, дасть FAST FULL SCAN. У цій ситуації, щоб ефективно запрацювали обидва індекси, потрібно вести додаткову умову для одного зі стовпців.

Індекс має великий фактор кластеризації CLUSTERING_FACTOR. По кожному індексу Oracle обчислює фактор кластеризації (ФК), що визначає число переміщень від одного блоку до іншого в таблиці при виборі індексом рядків з таблиці. Мінімальне значення ФК дорівнює числу блоків таблиці, максимальне – числу рядків в таблиці. Фактор кластеризації для індексу обчислюється під час збору статистики. Він використовується оптимізатором при розрахунку вартості індексного доступу до даних таблиці. Великий ФК (особливо близький до числа рядків в таблиці) говорить про неефективне використання індексу. Таким чином, ФК є характеристикою індексу, а не таблиці. Перше рішення при великому ФК є прибрати існуючий індекс як неефективний. Друге рішення, якщо даний індекс найбільш часто застосовується в запитах і він потрібен, то перебудувати структуру таблиці таким чином, щоб рядки в блоках таблиці були впорядковані в тому ж порядку, в якому розташована інформація за даними рядках в індексі, тобто зробити кластерни-

ми блоки таблиці, зменшивши таким чином кількість переміщень від одного блоку до іншого при роботі індексу [8].

Індекс давно не перебудовувався (індекс сильно фрагментований за рахунок багаторазових вилучень в таблиці). У цьому випадку може бути проведена або перебудова Rebuild індексів (здійснюється по команді ALTER INDEX owner. Ім'я індексу REBUILD ONLINE зі звільненням простору, в якому знаходитися індекс), або COALESCE – процедура зменшення числа листових блоків в індексі шляхом їх об'єднання без звільнення простору (ця процедура не блокує таблицю в процесі виконання і виконується за командою ALTER INDEX owner. ім'я індексу COALESCE).

Проблеми з хінтами в запиті. Проблеми з хінтами можуть бути наступні:

- неефективний хинт. Він може привести до істотного зниження продуктивності;
- хинт був написаний, коли БД працювала на 9-му Oracle, при переході на Oracle 10g і вище хинт став гальмом (це можуть бути хинти Rule, Leading і ін.).

Leading – потужний хинт, але при переході на іншу версію Oracle в деяких випадках призводить до різкого зниження продуктивності і перед застосування цих хинтов необхідно враховувати ймовірність зміни з часом статистики системи і її об'єктів (таблиць і індексів), які використовуються в запиті:

- у хинті USE_NL міститься не повний перелік алиасов;
- у складеному хинті використовується неправильний порядок проходження хинтів, в результаті чого хинти блокую ефективну роботу один одного. Наприклад, хинт Leading повністю ігноруються при використанні двох або більше конфліктуючих підказок Leading або при вказівці в ньому більше однієї таблиці;
- хинт написаний давно, після чого була модифікація запиту (наприклад, відсутня або змінився індекс, вказаний в хинти);
- у запиті відсутній хинт, який би підвищив ефективність роботи запиту. У ряді випадків наявність хинта підвищує ефективність запиту і забезпечує стабілізацію планів виконання (наприклад, при зміні статистики).

При створенні хинта в запиті є ряд рекомендацій:

- у хинті INDEX можуть бути пераховані кілька індексів. Оптимізатор сам вибере відповідний індекс. Можна поставити хинт NO_INDEX, якщо треба заблокувати використання якогось індексу;

– при наявності Distinct в запиті Distinct ставитися після хинта (тобто хинт завжди йде після Select);
– найбільш ефективні і часто використовуваними є хинти: Ordered, Leading, Index, No_Index, Index_FFS, Index_Join, Use_NL, Use_Hash, Use_Mer-ge, First_Rows (n), Parallel, Use_Concat, And_Equal, Hash_Aj і інші. При цьому, наприклад, індекс Index_FFS крім швидкого повного сканування індексу дозволяє йому виконуватися паралельно, в силу чого можна отримати суттєвий виграш в продуктивності.
– зміна параметрів ініціалізації бази даних в межах запиту дозволяє зробити хинт / * + opt_param ('Параметр ініціалізацці' N) * /, наприклад, / * + opt_param ('optimizer_index_caching' 10) * /. Даний хинт використовується для перевірки продуктивності роботи запиту в разі, коли запит розробляється або тестується на базі з одним значенням параметрів ініціалізації, а працює на базі з іншими значеннями.

Слід зазначити, що в деяких випадках, коли хинт неефективний, але замінити його оперативно в запиті не представляється можливим (наприклад, чужа розробка), є можливість, не змінюючи робочий запит в програмному модулі, замінити хинт (хинти) в запиті, а також в його підзапитах, на ефективний хинт (хинти) [3].

Це прийом – підміна хинтов (який відомий, як використання збережених шаблонів Stored Outlines). Але така підміна повинна бути тимчасовим рішенням до моменту коригування запиту, оскільки постійна підміна хинта може привести до деякого зниження продуктивності запиту.

2.3 Визначення причин неефективно складеного запиту

Причин неефективності запиту кілька:
– неефективне з'єднання таблиць;
– використання NOT і NOT IN в умови where;
– блокування індексу в силу використання неправильних функцій до колонку, за яким побудований індекс;
– вкладеність запиту або більша його довжина;
– великий обсяг обираних даних потребуючих підключення в роботу дисків, в тому числі для виконання агрегованих функцій (order by, group by);

– неефективні збережені процедури, які використовуються в запиті і ін.

Серед причин неефективності особливу увагу слід приділити неефективного з'єднанню таблиць (наявність HASH або MERGE з'єднань там, де краще NESTED LOOP – про що сказано вище). Крім того ефективність з'єднання може залежати від порядку таблиць у фразі FROM. Щоб оптимізатор працював з таблицями в тому порядку, в якому вони знаходяться у фразі From використовується хинт Ordered.

Ефективність з'єднання залежить від повноти зв'язку у фразі WHERE між таблицями. При недостатній зв'язці в плані виконання з'являється MERGE JOIN CARTESIAN.

Особлива увага при модифікації запиту слід приділити фразі NOT IN в умови where. Як варіант звільнення від NOT IN можна використовувати прийом, при якому пишеться перший запит без NOT IN, а за ним після MINUS пишеться той же запит з IN (віднімання з повного числа рядків рядка, одержувані після використання умови IN, який працює швидше, ніж NOT IN). З метою прискорення роботи запиту використовувати (там, де це можна) замість UNION фразу UNION ALL (UNION операція більш повільна, тому що здійснюється шляхом сортування).

Також рекомендується зменшувати число таблиць у фразі FROM. Це дозволить зробити план виконання прозорим для оптимізатора та його аналізу. В першу чергу прибрати з FROM таблиці, стовпці яких не використовуються після фрази Select. В цьому випадку можна використовувати вкладені запити з цими таблицями після Select або у фразі where [2].

Завдання діапазону дат, починаючи з 01.01.0001, призводить до неефективного плану виконання. Треба зробити мінімальну межу дати, тобто якомога ближче до реальної дати.

З метою підвищення продуктивності запиту не робити довгі запити, тому що довгий запит збільшує час розбору запиту оптимізатором, час передачі по каналах і займає надлишкову пам'ять.

З метою підвищення продуктивності роботи запиту ширше використовувати кешування всіх видів: послідовностей, таблиць, результатів виконання запитів. Кешування результатів виконання запитів з'явилося в Oracle 11g і дозволяє витягувати результат першого виконання запиту з оперативної пам'яті. Це особливо ефективно при великому числі виконання запиту і відсутність в момент багаторазового виконання запиту операцій DML над таблицею [5].

3 ВИБІР ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ ПІДСИСТЕМИ

Для здійснення реалізації підсистеми дослідження та аналізу алгоритмів оптимізації виконання SQL-запитів треба обрати технології та програмні засоби реалізації підсистеми.

Під час аналізу існуючих мов і порівняння їх можливостей, вибір зупинився на мові програмування Java. Використання даної мови надає можливість реалізувати UI стандартними засобами Swing. В якості системи управління базою даних була обрана Oracle. Реалізації

3.1 Система управління базою даних Oracle

В даний час для побудови інформаційних систем застосовуються різні системи управління базами даних (СУБД), що розрізняються як своїми можливостями, так і вимогами до обчислювальних ресурсів. Все різноманіття застосовуваних СУБД, проте, можна звести до двох основних їх класам: персональні і розраховані на багато користувачів.

До першого класу відносяться СУБД, орієнтовані для роботи на персональному комп'ютері (dBASE, FoxPro, MS Access і т.п.). Спочатку вони підтримували роботу з даними тільки одного користувача. Вся СУБД такого класу виконується як одна програма, таблиці бази даних представляються окремими файлами на диску того ж персонального комп'ютера. З розвитком локальних мереж розробники СУБД цього класу стали пристосаблювати їх до роботи в мережевому середовищі, в якій потенційно стало можливим організувати доступ до даних з декількох персональних комп'ютерів, включених в локальну мережу. Файли бази даних при цьому розміщуються на файловому сервері. На кожному ж робочому місці працює власна копія програми-СУБД і прикладна програма, і на їх виконання можуть надавати значний вплив характеристики комп'ютера цього робочого міста [8].

Таким чином, при наявності в мережі N робочих місць з одними і тими ж даними працюють N копій програми-СУБД, одними і тими ж даними управляють відразу N копій СУБД. Помилка у виконанні однієї з копії є буде помічена іншими копіями. При виконанні запитів до бази даних копія СУБД може або виробляти пошук даних у віддалених файлах на файловому сервері, або копіювати всі файли, в яких ведеться пошук в свою локальну файлову систему. У першому випадку виникають проблеми одночасного доступу до даних при їх зміні.

Дані, над якими проводиться зміни, повинні бути заблоковані. Засоби файлового серверу дозволяють виконувати блокування на рівні файлів, але не на рівні записів, що суттєво знижує ефективність паралельної роботи з базою даних багатьох користувачів. У другому ж випадку, по-перше, потрібна передача по мережі великих обсягів інформації, а по-друге, виходить, що різні робочі місця працюють з різними копіями даних і ці копії можуть стати неідентичних.

СУБД другого класу такі як Oracle і Microsoft SQL Server спочатку створювалися для виконання на великих комп'ютерах і забезпечення паралельної роботи багатьох користувачів. Такі СУБД, як правило, складаються з ядра, постійно присутнього в пам'яті, (сервера) і великої кількості програм-агентів, які обслуговують запити кінцевих користувачів і прикладних програм (клієнтів). В цьому випадку і ядро СУБД, і дані знаходяться на одному і тому ж комп'ютері. Одна копія СУБД управляє однією копією даних. Єдина керуюча система дозволяє ефективно організувати одночасний доступ до даних багатьох агентів, запобігаючи конфлікти між ними. Помилка в роботі СУБД локалізована і може бути ефективно виправлена самої ж СУБД. При роботі в умовах мережі ядро СУБД виконує запити агентів на вибірку даних і передає по мережі тільки результати вибірки. Оскільки швидкодія сучасних дискових систем зазвичай вище, ніж швидкість передачі даних по мережі, зменшення обсягу переданих даних істотно збільшує загальну ефективність роботи системи. При цьому не накладається ніяких обмежень на масштаб мережі, агенти можуть бути пов'язані з ядром СУБД через любу мережу і будь-які протоколи передачі даних. [9]

Багаториствувальницьки СУБД володіють також незаперечними перевагами в таких аспектах, як надійність, безпеку, доступність. Багопользовательскіе СУБД з самого початку своєї історії використано в якості інтерфейсу запитів мову SQL, звідси відбуло одне з їх альтернативних назв – SQL-сервери. Хоча в останній час підмножини SQL стають доступними і в персональних СУБД, але в ці підмножини не включаються кошти забезпечення безпеки і паралельного доступу до даних – ті кошти, які персональні СУБД забезпечити просто невзмозі.

Oracle займає лідируючі позиції на ринку СУБД і, що особливо важливо, лідирує на платформах Unix і Windows. Причина широкої поширеністі Oracle полягає насамперед у високих експлуатаційні характеристики СУБД, великої кількістю підготовлених вітчизняних фахівців з Oracle, наявністю підтримуючої інфраструктри – навчальних центрів, широкої мережі партне-

рів Oracle, великому числу технічних курсів по Oracle в вищих навчальних закладах і т.д

З технічної точки зору важливим є те, що Oracle функціонує практично на всіх платформах. Іншою важливою характеристикою являєтся підтримка Oracle всіх можливих варіантів архітектур, в тому числі симетричних багато-процесорних систем, кластерів, систем з масовим паралелізмом і т.д.

Очевидна значимість цих характеристик для систем масштабу корпорації, де експлуатуються безліч комп'ютерів різних моделей і робите-лей. В таких умовах фактором успіху є максимально можлива типізація пропонованих рішень, яка має за мету істотне зниження вартості володіння програмним забезпеченням. Уніфікація систем управління базами даних – один з найбільш значущих кроків на шляху досягнення цієї мети. Ядром СУБД Oracle є сервер бази даних, який поставляється в одному з чотирьох варіантів в залежності від масштабу інформаційної системи, в рамках якої передбачається його застусування. Для систем масштабу великої організації пропонується продукт Oracle Database Enterprise Edition (корпоративна редакція), для якого є цілий набір опцій, архітектурно і функціонально розширюють можливості сервера. Саме Oracle Database Enterprise Edition встановлюється на кластерах (з опцією Parellel Server), дозволяючи створювати системи високої готовності. Продукт Oracle Database Standard Edition (стандартна редакція) орієнтований на організацію середнього масштабу або підрозділу в складі великої організації. Для персонального використання призначений продукт Oracle Database Personal Edition (персональна редакція) [6].

Найважливішою перевагою Oracle перед конкурентами (і, перш за все, перед SQL Server) є ідентичність коду (в оцінці Gartner Group – консолідація коду) різних версій сервера баз даних Oracle для всіх платформ, що гарантує ідентичність і передбачуваність роботи Oracle на всіх типах комп'ютерів, які б не входили до її складу.

Всі варіанти сервера Oracle мають в своїй основі один і той же вихідний програмний код і функціонально ідентичні, за винятком деяких опцій, які, наприклад, можуть бути додані до Oracle Database Enterprise Edition і не можуть – до Oracle Database Standard Edition. Таким чином, для всіх платформ існує єдина СУБД в різних версіях, яка поводиться однаково і надає однакову функціональність незалежно від платформи, на якій вона встановлена.

Розробку серверних продуктів в складі СУБД виконує єдиний підрозділ корпорації Oracle, зміни вносяться централізовано, після цього піддаються

ретельному тестуванню в базовому варіанті, а потім переносяться на всі платформи, де також детально перевіряються. Можливість перенесення Oracle забезпечується специфічною структурою вихідного програмного коду сервера. Приблизно 80% програмного коду Oracle – це програми на мові програмування C, яка (з відомими обмеженнями) є незалежною від платформ. Приблизно 20% коду, що представляє собою ядро сервера, реалізовано на машинно-залежних мовах і ця частина коду, зрозуміло, переписується для різних платформ

Жорстка технологічна схема розробки Oracle, яка спиралася на принципи ідентичності вихідного програмного коду для різних версій і платформ, контрастує зі схемами других компаній. Отже, СУБД Oracle приховує деталі реалізації механизмов управління даними на кожній з платформ, що дає змогу говорити про практично повної уніфікації базового програмного забезпечення. Додатково до цього, архітектура Oracle дозволяє переносити прикладні системи, реалізовані на одній платформі, на інші платформи без змін як в структурах баз даних, так і у кодах додатків. При цьому основним критерієм, що визначає можливість переносу тих чи інших програмних компонентів між платформами є повне виключення з них машинно-залежного коду.

СУБД Oracle має унікальні якостями переносимості, а також надає відкриту платформу для розробки переносите додатків клієнт / сервер і Internet / Intranet-додатків.

Наявність декількох редакцій сервера баз даних – корпоративної, стандартної, персональної і повна переносимість додатків між ними дозволяє задовольнити потреби муніципальної інформаційної системи і кардинально вирішити задачу уніфікації базового програмного забезпечення [1].

Пакет Oracle, наділений найрозвиненішим набором функцій для роботи з мовою Java і доступу до даних через Інтернет, системою оптимізації одночасного доступу. Єдиним недоліком даної СУБД є складність адміністрування, однак усі витрати на її впровадження та освоєння надалі повернуться ефективною і надійною роботою.

Oracle Database 11g XE (випущена в 2007 році) – це об'єктно–реляційна система керування базами даних компанії Oracle.

Багато засобів автоматичної настройки та управління отримали подальший розвиток, особливо Automatic Memory Management, секціонування і механізми забезпечення безпеки. Усередині Oracle Enterprise Manager розширено життєвий цикл управління змінами СУБД, оскільки тепер Oracle надає поліпшені можливості діагностики і підключення до служби технічної підтрим-

ки Oracle Support за допомогою підсистеми Support Workbench. Ця версія може також похвалитися вдосконаленими засобами онлайнового накладення «латок». Ця СУБД початкового рівня, доступна для Windows і Linux безкоштовно. Може використовувати не більше 1 Гбайт пам'яті і 4 Гбайт дискового простору. Надає частину функціональності, включеної до редакції Standard Edition One. Відсутні такі функції, як віртуальна Java–машина, кероване сервером резервне копіювання і відновлення, а також підсистема Automatic Storage Management. Oracle Enterprise Manager не вміє управляти цією СУБД, проте її можна розгорнути так, що вона буде доступна з адміністративного інтерфейсу Oracle Application Express (колишній HTML-DB), що дозволяє керувати кількома користувачами [8].

3.2 Мова програмування Java

Одне з головних переваг мови Java – його незалежність від платформи, на якій виконуються програми. Таким чином, один і той же код можна запускати під управлінням операційних систем Windows, Linux, FreeBSD, Solaris, Apple Mac і ін. Це стає дуже важливим, коли програми завантажуються за допомогою глобальної мережі інтернет і використовуються на різних платформах. Java – це одночасно мова програмування і платформа.

Java є високорівневою об'єктно-орієнтованою мовою програмування. При компіляції, яка виконується один раз під час збірки додатку, код на Java перетворюється в код проміжною мовою (байт-код). У свою чергу, байт-код аналізується і виконується (інтерпретується) віртуальною машиною Java (JVM), яка грає роль транслятора між мовою Java і апаратним забезпеченням з операційною системою. Все реалізації Java повинні емулювати JVM, щоб створювані додатки могли виконуватися на будь-якій системі, що включає віртуальну машину Java [10].

По–друге, Java – це програмна платформа, версії якої поставляються для різних апаратних систем. Платформа включає в себе JVM і інтерфейс прикладного програмування на Java (API), що являє собою великий набір готових програмних компонентів (класів), що полегшують розробку і розгортання аплетів і додатків. API Java охоплює багато аспектів розробки на Java, в тому числі маніпулювання базовими об'єктами, мережеве програмування, забезпечення безпеки, генерацію XML і Web-сервіси. API організований у вигляді набору бібліотек, іменованих пакетами, які містять класи і інтерфейси для вирішення пов'язаних один з одним завдань.

На додаток до API кожна повноцінна реалізація платформи Java містить:

- інструментарій розробника для компіляції, запуску, моніторингу, налагодження та документування додатків;
- стандартні механізми розгортання додатків в призначеній для користувача середовищі;
- інструментарії, що дозволяють створювати складні графічні інтерфейси користувачів;
- інтеграційні бібліотеки для програмного доступу до баз даних і віддаленого маніпулювання об'єктами.

JVM також є перевіреним середовищем для виконання додатків, написаних на відмінних від Java мовами. Зокрема, Groovy, Scala і спеціалізовані реалізації Ruby і Python надають розробникам можливість виконання на JVM динамічних і функціональних мов [11].

3.3 Бібліотека Swing

Попередня оптимізація та тестування алгоритмів оптимізації буде виконуватися напряму в середовищі Oracle, призначення майбутньої програми – відображення процесу виконання SQL-запитів та планів виконання цих запитів, тобто представлення результатів аналізу алгоритмів в наочній та приємній формі, чого складно досягти в стандартних середовищах Oracle, таких як SQL*Plus та SQL–developer, тому небхідно, щоб вона мала графічний інтер-фейс, для цієї мети було обрано Java – фреймворк – Swing [9].

Swing – бібліотека для створення графічного інтерфейсу для програм на мові Java. Свінг був розроблений компанією Sun Microsystems.

Архітектура Гойдалки розроблена таким чином, що ви можете змінювати «виглядати і відчувати себе» (L & F) прикладної програми. «Look» визначає зовнішній вигляд компонентів, а «Feel» – їх поведінка. Компанії Sun JRE надає наступні L & F.

CrossPlatformLookAndFeel – це рідний L & F для Java-додатків (так само називається Metal). Він використовується за умовчанням, забезпечуючи стандартну поведінку компонентів і їх зовнішній вигляд, незалежно від платформи, на якій запускається додаток [12].

SystemLookAndFeel – в цьому випадку додаток використовує L & F, який є рідним для системи, на якій запущено програму. Системний L & F визначається під час виконання. Для Windows використовується «Windows» L

& F, який імітує особливості конкретної системи, на якій запущений – класичний Windows, XP, або Vista. Для Linux і Solaris використовується «GTK +», якщо GTK + Встановлено 2.2 або більш пізня версія, в іншому випадку використовується «Мотиву».

Synth – основа для створення власних L & F.

Multiplexing – надає можливість використання різних L & F одночасно.

3.4 Автоматизация процесса сборки

Певний процес складання це один з найважливіших, але не повсеме стно використовуваних елементів розробки програмного забезпечення. За своєю природою це накладне заняття, тісно пов`язане з розробкою. Визналенний процес складання гарантує, що при запуску збірки розробляемого проекту кожен раз проробляються одні і ті ж дії. З ускладнитиням процесу складання, наприклад при розробці EJB додатків або схожих за складністю, стає вкрай необхідно визначити певний стандарт збірки. Ви повинні якомога точніше визначити, інвентаризують-вать і автоматизувати точний набір певних кроків [10].

Певний процес складання – важлива складова будь-якого циклу розробки, оскільки дозволяє скоротити нестиковки між різними стадіями розробки якось: написання коду, інтеграція, тестування, продаж. Наявність певного процесу складання дозволить прискорити перехід з однієї стадії в іншу. Також зникають різного роду проблеми пов'язані з компіляцією, змінної classpath, які в сукупності могли б бути причиною розтрати в порожню безлічі часу і грошей.

Apache Ant – утиліта для автоматизації процесу збирання програмного продукту. Є платформонезавісимость аналогом утиліти make, де всі команди записуються в XML-форматі.

Перша версія була розроблена інженером Sun Microsystems Джеймсом Девідсоном (James Davidson) який потребував утиліті, подібної make.

Ant, на відміну від іншого збирача проектів Apache Maven, забезпечує імперативну, а не декларативну збірку проекту.

На відміну від make, утиліта Ant повністю незалежна від платформи, потрібно лише наявність на застосовуваної системі встановленої робочого середовища Java – JRE. Відмова від використання команд операційної системи і формат XML забезпечують переносимість сценаріїв.

Управління процесом складання відбувається за допомогою XML-сценарію, який також називають Build-файлом. В першу чергу цей файл містить визначення проекту, що складається з окремих цілей (Targets). Цілі можна порівняти з процедурами в мовах програмування і містять виклики команд-завдань (Tasks). Кожне завдання являє собою неподільну, атомарному команду, що виконує деякий елементарне дію [10, 11].

Між цілями можуть бути визначені залежності – кожна мета виконується тільки після того, як виконані всі цілі, від яких вона залежить (якщо вони вже були виконані раніше, повторного виконання не проводиться).

Типовими прикладами цілей є clean (видалення проміжних файлів), compile (компіляція всіх класів), deploy (розгортання програми на сервері). Конкретний набір цілей і їх взаємозв'язку залежать від специфіки проекту.

Ant дозволяє визначати власні типи завдань шляхом створення Java-класів, що реалізують певні інтерфейси.

4 ПРОЕКТУВАННЯ ПІДСИСТЕМИ АНАЛІЗУ АЛГОРИТМІВ ОПТИМІЗАЦІЇ

Перед розробкою підсистеми необхідно чітко уявляти, які функціональні можливості будуть закладені в систему і як буде організовано функціональне взаємодія всередині системи.

При розробці функціональної моделі може виникнути безліч проблем.

При її розробці спочатку будується модель існуючої організації роботи AS-IS (як є) на основі посадових інструкцій, наказів, звітів, нормативної документації тощо Вона дозволяє з'ясувати, «що ми робимо сьогодні» перед тим, як «перестрибнути» на те, «що ми будемо робити завтра» Аналіз моделі дозволяє зрозуміти, де знаходяться слабкі місця, в чому полягатимуть переваги нових процесів і наскільки глибоким змінам піддасться [13].

Знайдені в моделі недоліки виправляються при створенні моделі TO-BE (як буде). Модель TO-BE потрібна для аналізу альтернативних шляхів вирішення завдання і вибору найкращого з них.

Слід вказати на поширену помилку при створенні моделі TO-BE – це створення моделі, що ідеалізується. Прикладом може служити створення моделі на основі знань керівника, а не конкретного виконавця робіт. Керівник знайомий з тим, як передбачається виконання роботи по посібникам і посадових інструкцій і часто не знає, як насправді підлеглі виконують роботи. В результаті виходить прикрашена, перекручена модель, яка несе неправдиву інформацію і яку неможливо надалі використовувати для аналізу. Така модель називається SHOULD-BE (як повинно було бути).

В даний час двома найбільш популярними методологіями побудови функціональних моделей є SADT і DFD.

4.1 Проектування підсистеми за допомогою методології функціонального моделювання SADT

Методологія SADT являє собою сукупність методів, правил і процедур, призначених для побудови функціональної моделі об'єкта будь-якої предметної області. IDEF0 – методологія та стандарт функціонального моделювання. За допомогою графічної мови IDEF0, інформаційна система постає у вигляді набору взаємопов'язаних функціональних блоків. Моделювання засобами IDEF0, як правило, є першим етапом вивчення системи [14].

При проектування інформаційної системи була обрана методологія функціонального моделювання SADT (Structured Analysis and Design Technique) – (стандарт IDEF0).

Контекстна діаграма є вершиною деревовидної структури діаграм і являє собою саме загальний опис системи та її взаємодія з зовнішнім середовищем. Проектування починається з представлення системи як єдиного цілого – одного функціонального блоку з граничними стрілками, які простираються за межі аналізованої області.

Контекстна діаграма підсистеми дослідження аналізу алгоритмів оптимізації виконання SQL-запитів наведена на рисунку 4.1.

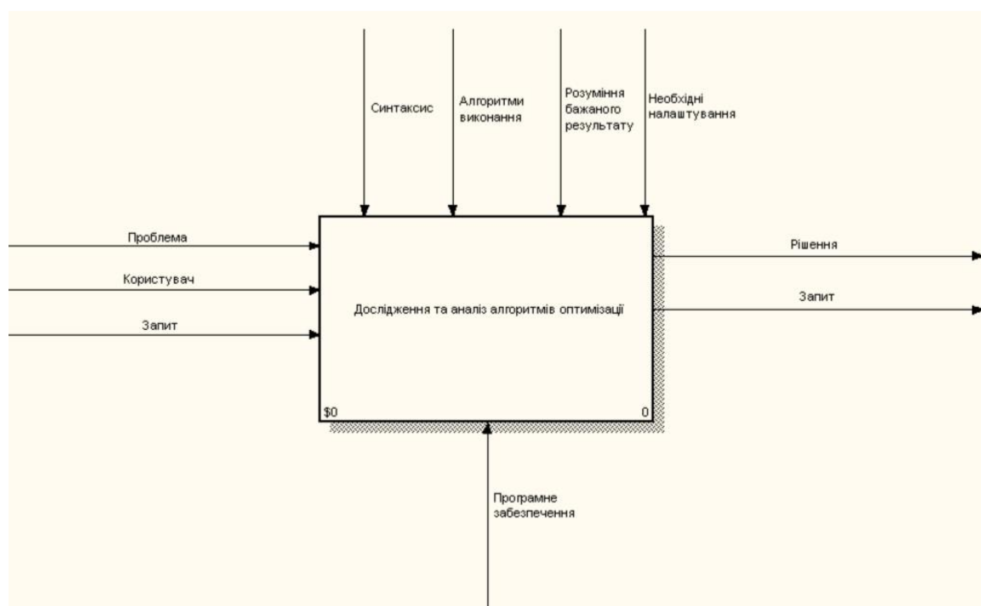

Рисунок 4.1 – Контекстна діаграма підсистеми

На контекстній діаграмі відображена головна робота підсистеми « Дослідження та аналіз алгоритмів оптимізації». На вхід подається інформація про член суспільства і запит чи інформація яку треба відредагувати. Головна робота керується: алгоритмом виконання, розуміння бажаного результат та необхідними налаштунками підсистеми .

Після опису системи в цілому проводиться розбиття її на великі фрагменти. Цей процес називається функціональна декомпозиція, а діаграми, які описують кожен фрагмент і взаємодію фрагментів, називаються діаграмами декомпозиції .

Після декомпозиції контекстної діаграми проводиться декомпозиція кожного великого фрагмента системи на більш дрібні і так далі, до досягнення потрібного рівня подробиці опису. Після кожного сеансу декомпозиції

проводяться сеанси експертизи – експерти предметної області вказують на відповідність реальних процесів створеним діаграмам. Знайдені невідповідності виправляються, і тільки після проходження експертизи без зауважень можна приступати до наступного сеансу декомпозиції. Так досягається відповідність моделі реальним процесам на кожному рівні декомпозиції моделі. Синтаксис опису системи в цілому і кожного її фрагмента однаковий у всій моделі.

Після декомпозиції контекстної діаграми отримуємо 3 блоку – роботи. Ці блоки представляють основні під функції початкової функції.

Функція «Розробка підсистеми» включає в себе повну розробку інформаційної системи на локальному комп'ютері. Включає в себе розробку інтерфейсу, скриптів. Входом у неї є дані о проблемі. Управляється за допомогою синтаксиса. Механізмом є програмне забезпечення, яке потрібне для розробки. І результатом роботи є готова підсистема.

Функція «Налаштування підсистеми» служить для можливості отримати доступу на кінцевому компьютері. Входом для роботи є готова підсистема для розміщення. Управляється робота налаштуваннями. Механізмом є – програмне забезпечення.

Функція «Пошук рішення» призначена для того, щоб проаналізувати та найти ріщення. Наприклад, це: виконання алгоритмів. У даної роботи є чотири входи, це налаштована підсистема, проблема, користувач та сам запит. Управляється правилами розуміння бажаного результату. Механізмом є – програмне забезпечення. Виходом даної роботи є найдене запит та само рішення, що запит отптимизований чи ні. Діаграма декомпозиції наведена на рис. 4.2.

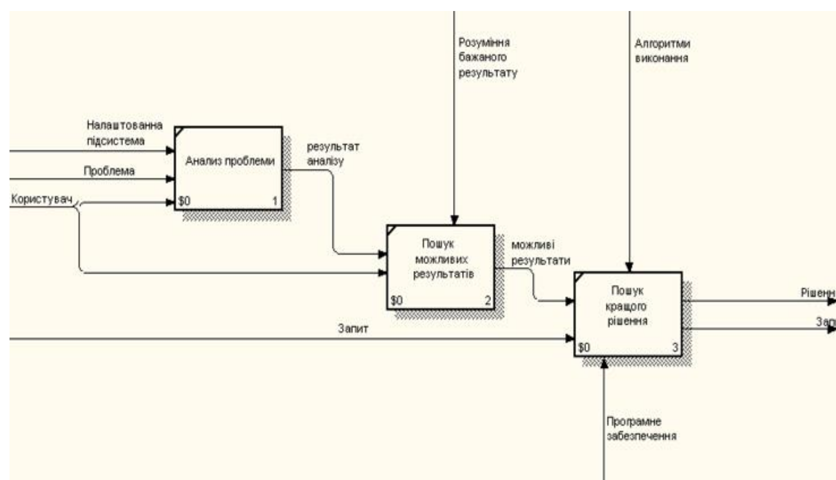

Рисунок 4.2 – Діаграма декомпозиції підсистеми

При декомпозиції першого А1 блоку виділяються наступні блоки:

«Розробка функціоналу» –вхід у даної роботи є проблема, управління синтаксисом, механізм – програмне забезпечення, виходом є розроблений функціонал.

«Створення інтерфейсу» – вхід у даної роботи є розроблений функціонал, управління – синтаксис, механізм – програмне забезпечення, вихід – функціонал з інтерфейсом.

«Динамічне підключення» – вхід у даної роботи є функціонал з інтерфейсом, управління – синтаксис, механізм – програмне забезпечення, вихід – розроблена підсистема.

Діаграма декомпозиції блоку А1 представлена на рисунку 4.3.

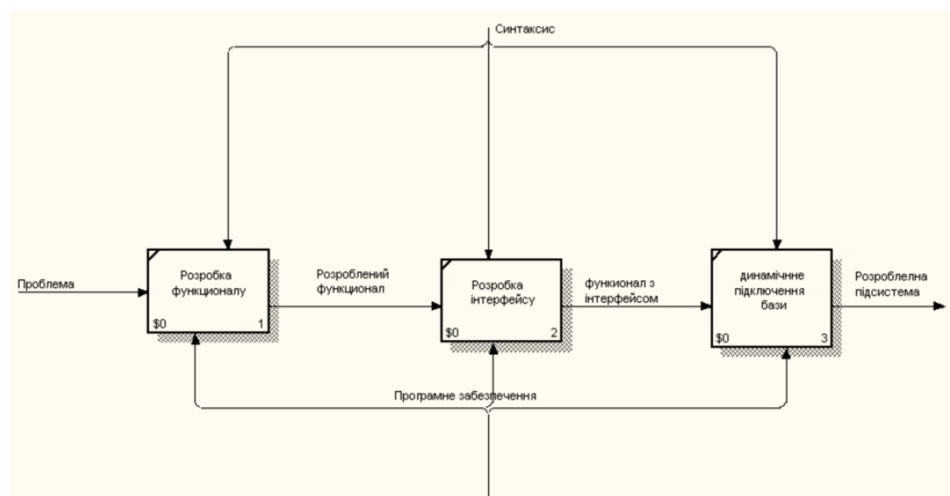

Рисунок 4.3 – Діаграми декомпозиції першого А1 блоку

При декомпозиції другого А2 блоку виділені 3 роботи:

«Установка оточення для підсистеми» – вхід у даної є роботи розроблена підсистема та користувач, управління – необхідними налаштуваннями, вихід – оточення для підсистеми.

«Установка та налаштування бази даних» – вхід у даної є роботи розроблена підсистема та користувач, управління – необхідним їм налаштуваннями, вихід – налаштована база.

«Підключення підсистеми до бази даних» – вхід у даної є роботи розроблена підсистема та налаштована база, механізм – програмне забезпечення, вихід – налаштована підсистема.

Діаграма декомпозиції блоку А2 представлена на рисунку 4.4.

При декомпозиції третього А3 блоку виділені 3 робіт:

«Аналіз проблеми» – для аналізу проблеми потрібно: налаштована підсистема, проблема, користувач та сам запит, механізм – це програмне забезпечення; результатом роботи є результат аналізу користувач.

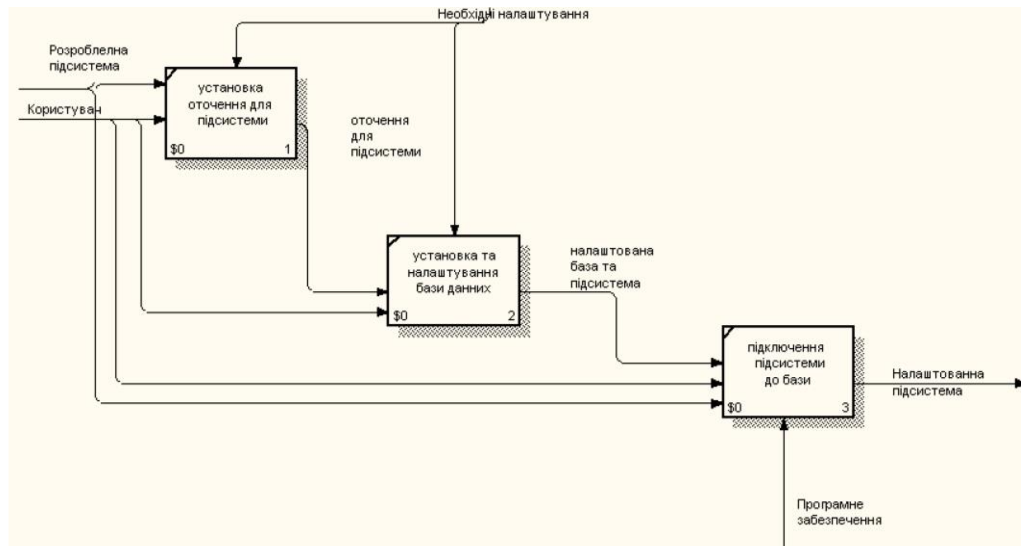

Рисунок 4.4 – Діаграми декомпозиції другого А2 блоку

«Пошук можливих рішень» – щоб організувати пошук можливих рішень потрібно мати аналіз результату та користувача; управлінням є розуміння бажаного результату; механізмом є програмне забезпечення, а результатом є – можливі результати.

«Пошук кращого рішення» – щоб організувати пошук кращого рішення потрібно мати можливе рішення та запит ; управлінням алгоритм виконання; механізмом є програмне забезпечення, а результатом рішення та сам запит. Діаграма декомпозиції третього А3 блоку представлена на рис. 4.5.

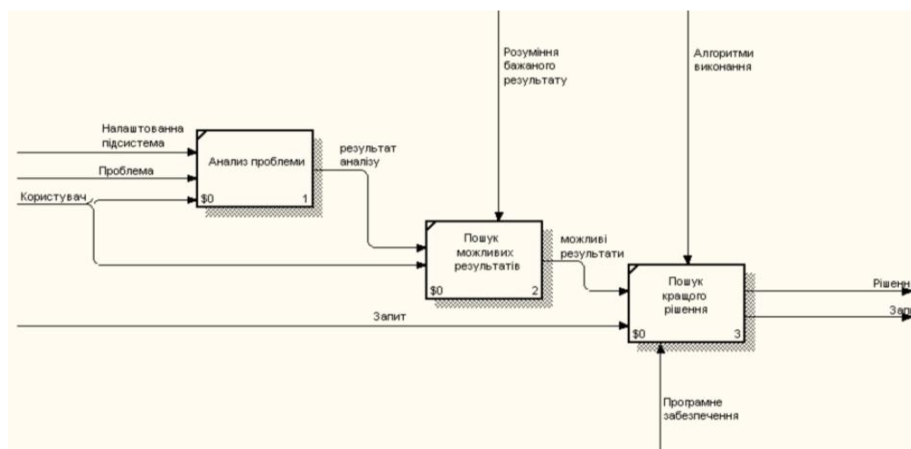

Рисунок 4.5 – Діаграми декомпозиції третього А3 блоку

4.2 Проектування підсистеми за допомогою послідовного виконання процесів Workflow Diagramming

Проектування інформаційно системи було здійснено за допомогою послідовного виконання процесів Workflow Diagramming (Стандарт IDEF3).

За допомогою IDEF3 описується логіка виконання дій. IDEF3 може використовуватися самостійно і спільно з методологією IDEF0: будь-який функціональний блок IDEF0 може бути представлений у вигляді послідовності процесів або операцій засобами IDEF3.

Якщо IDEF0 описує, що робиться в системі, то IDEF3 описує, як це робиться.

Контекстна діаграма в IDEF3 відображає основну функцію системи. Вона складається з єдиної роботи – «Дослідження та аналіз алгоритмів оптимізації». Контекстна діаграма представлена на рис. 4.6.

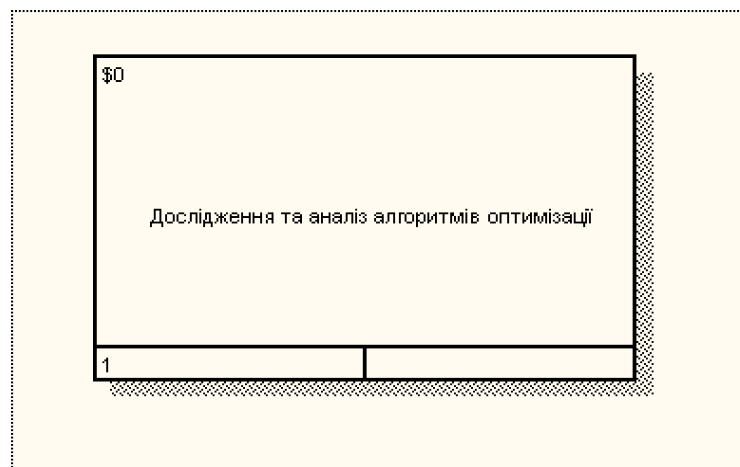

Рисунок 4.6 – Контекстна діаграма підсистеми

Провівши декомпозицію контекстної діаграми, спостерігається послідовність виконання робіт.

Першою роботою системи є «Розробка підсистеми». Після неї йде робота під назвою «Налаштування підсистеми» в якій пов'язані старшим зв'язком.

Далі робота «Пошук рішення» пов'язана з блоком « Налаштування підсистеми » також старшим зв'язком, означає те, що всі попередні роботи повинні, завершиться для того, щоб можна було без перешкод продавати товари.

Діаграма декомпозиції, представлена на рис. 4.7.

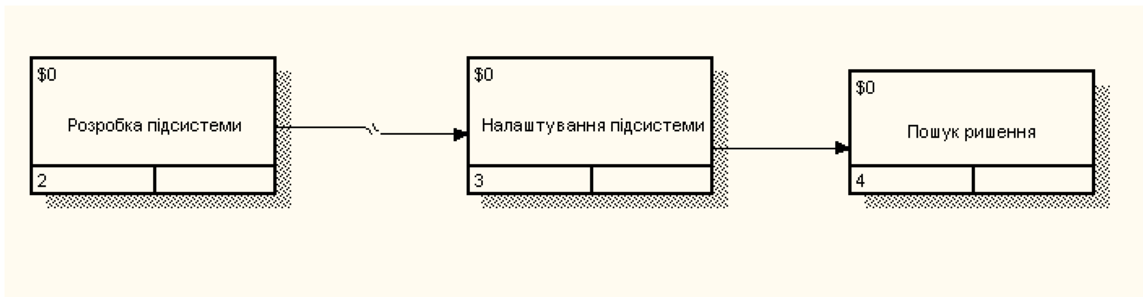

Рисунок 4.7 – Діаграма декомпозиції підсистеми

При декомпозиції наступного рівня роботи «Розробка підсистеми» отримали три блоки – це роботи з двома перехрестями. Діаграма декомпозиції, представлена на рисунку 4.8.

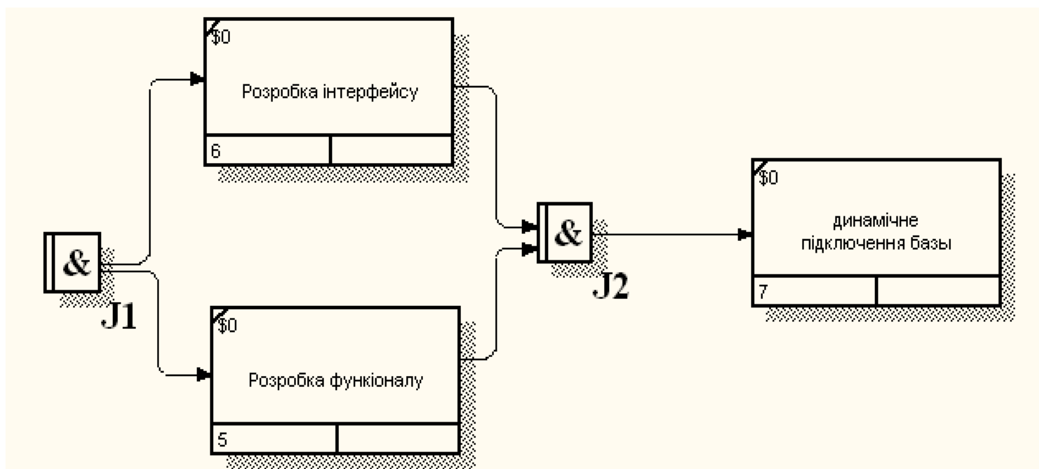

Рисунок 4.8 – Діаграма декомпозиції роботи «Розробка підсистеми»

Перше перехрестя «Асинхронне І», означає, що подальші роботи можуть початися не одночасно, але обов'язково повинні бути запущені. Це такі роботи як: «Розробка функціоналу» та «Створення інтерфейсу». При злитті стрілок-виходів з цих робіт, використовується таке ж перехрестя «Асинхронне І». Означає те, що роботи мають, завершиться, але це може бути не одночасно. Далі перехрестя і робота «Динамічного підключення бази».

При декомпозиції наступного рівня роботи «Налаштування підсистеми» отримали три блоки – це роботи з двома перехрестями. Діаграма декомпозиції, представлена на рисунку 4.9. Це такі роботи як: «Установка оточення для підсистеми» та «Установка та налаштування базі даних». При злитті стрілок-виходів з цих робіт, використовується таке ж перехрестя «Асинхронне І». Означає те, що роботи мають, завершиться, але це може бути не одночасно. Далі перехрестя і робота «Підключення підсистеми до бази».

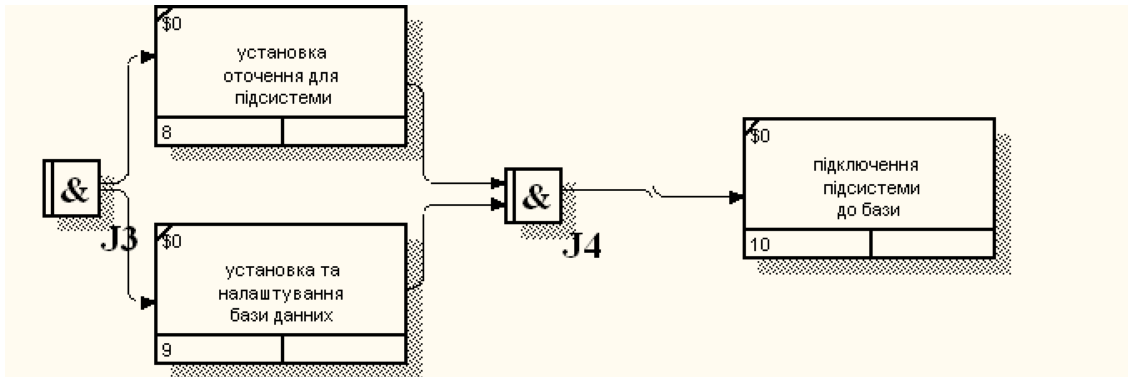

Рисунок 4.9 – Діаграма декомпозиції роботи «Налаштування підсистеми»

При декомпозиції роботи «Пошук рішення» всі роботи пов'язані старшим зв'язком і розміщені послідовно. А саме: «Аналіз проблеми», «Пошук можливих результатів», « Пошук кращого рішення». Діаграма декомпозиції, представлена на рисунку 4.10.

Рисунок 4.10 – Діаграма декомпозиції роботи «Пошук рішення»

4.3 Проектування підсистеми за допомогою діаграми потоків даних DFD

У відповідність з розглянутими методологіями модель аналізованої підсистеми дослідження аналізу алгоритмів оптимізації виконання SQL-запитів визначається як ієрархія діаграм потоків даних DFD, що описують процес перетворення інформації від введення в систему до видачі інформації адміністратору.

Діаграми потоків даних використовуються для опису руху документів і обробки інформації як додаток до методології функціонального моделювання IDEF0. На відміну від методології IDEF0, стрілки на діаграмах DFD показують лише те, як об'єкти (включаючи дані) рухаються від однієї роботи до іншої. Діаграма потоків даних DFD – це граф, на якому показано рух значень

даних від їх джерел через перетворюючи їх процеси до їх споживачів в інших об'єктах.

Діаграми верхніх рівнів ієрархії (контекстні діаграми) відображають зв'язок основного процесу системи із зовнішніми сутностями, які визначаються відповідними входами і виходами. Контекстні діаграми деталізуються за допомогою діаграм нижнього рівня.

У контекстній діаграмі головним процесом системи є «Покращення роботи запиту». Зовнішніми сутностями, які впливаю на систему, є: «Сервер» і «Користувач» . Існує блок як сховище даних. Зв'язок між користувачем і головною роботою полягає в «Зменшити час роботи запиту». Із системи йдуть дані в зовнішні сутності «Сервер» – надає доступ до підсистеми аналізу. Зі сховища даних в систему передаються дані про члена організації , діаграма зображена на рисунку 4.11 .

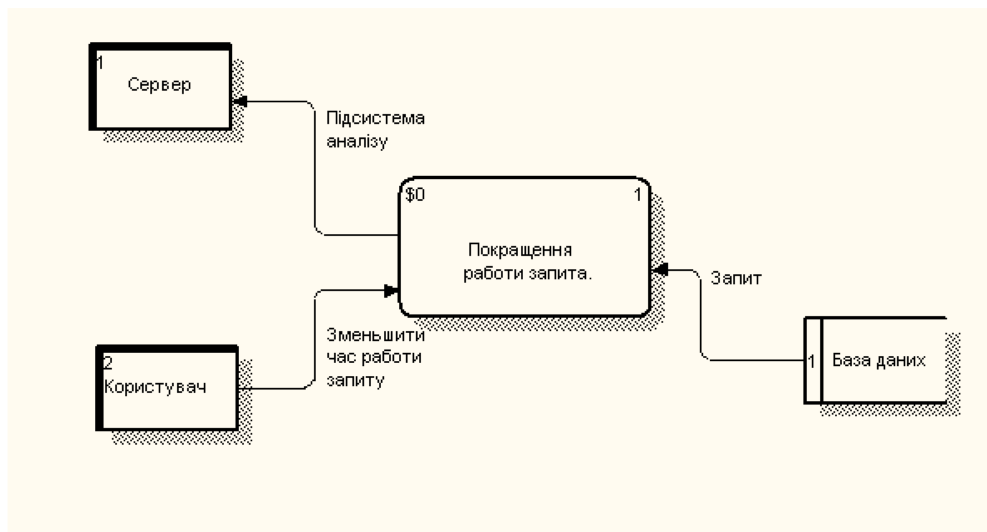

Рисунок 4.11 – Контекстна діаграма підсистеми

5 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПІДСИСТЕМИ

5.1 Розробка інтерфейсу підсистеми

Для реалізації підсистеми дослідження та аналізу алгоритмів оптимізації виконання SQL-запитів реалізоване програмне середовище, яке здійснює підключення до СУБД та план виконання запиту. В програмі були спроектовані і реалізовані всі необхідні елементи інтерфейсу, які зображені на рис. 5.1.

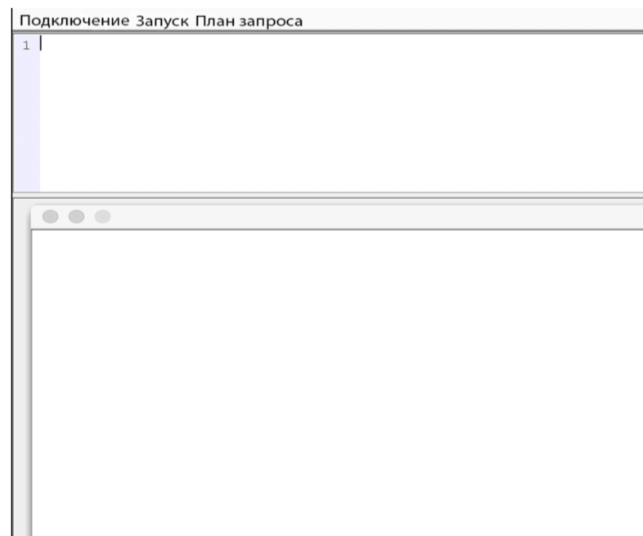

Рисунок 5.1 – Головне вікно підсистеми дослідження та аналізу алгоритмів оптимізації виконання запитів

Для виконання запитів користувачеві необхідно ввести свій запит до спеціального вікна зображене на рисунку 5.2.

```
1 select c.country_name,(COUNT(e.last_name)) as count
2 from countries c,employees e,departments d,locations l
3 where c.country_id = l.country_id
4 and l.location_id = d.location_id
5 and d.department_id = e.department_id
6 group by c.country_name;
7
```

Рисунок 5.2 –Вікно підсистеми для введення запиту

Підсистема підсвічує основні команди мови SQL. Що приводити в більш читабельного виду і легше орієнтуватися в коді. Процес запуску запиту

відбувається після натискання кнопки «Запуск». Підсистема обробляє запит і відправляє в базу. База після обробки запиту повертає результат. Приклад результату зображений на рисунку 5.3.

	COUNTRY_NAME	COUNT
1	United Kingdom	75045
2	United States of America	821523
3	Germany	37193
4	Canada	37345

Рисунок 5.3 – Вікно підсистеми для виведення результату запиту

Підсистема дозволяє динамічне підключення до бази даних. Дана вимога є обов'язковою для систем подібного типу. Можливість використання функціоналу динамічного підключення в підсистемі відбувається після натискання кнопки «Підключення». Дану дію викликає спливаюче вікно в якому необхідно заповнити поля для створення підключення до бази (рис. 5.4).

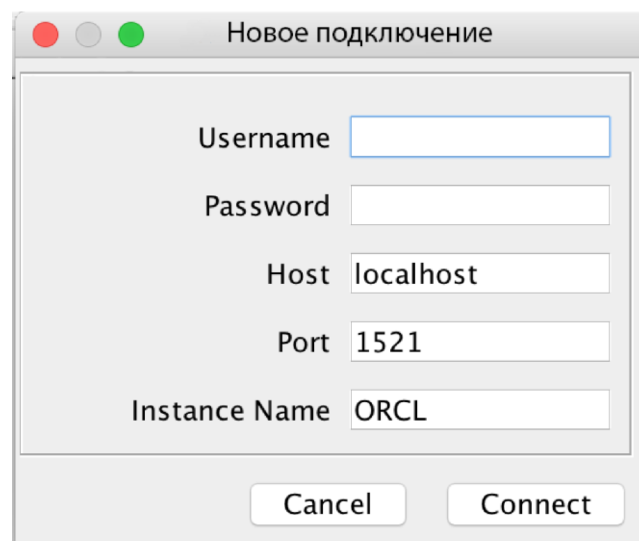

Рисунок 5.4 – Вікно підсистеми для створення нового підключення до бази даних

Користувач може отримати план запиту вказаного у вікні запиту, використовуючи кнопку «План запиту».

Підсистема автоматично оберне введений запит і дозволить користувачеві без додаткових втручань в код, отримати його план. Даний план надасть можливість проаналізувати запит з більш детальною інформацією (рис 5.5).

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                   |    14 |   392 |   642  (16)| 00:00:08 |
|   1 |  HASH GROUP BY             |                   |    14 |   392 |   642  (16)| 00:00:08 |
|   2 |   NESTED LOOPS             |                   |    14 |   392 |   641  (16)| 00:00:08 |
|   3 |    VIEW                    | VW_GBF_25         |    14 |   224 |   641  (16)| 00:00:08 |
|   4 |     HASH GROUP BY          |                   |    14 |   224 |   641  (16)| 00:00:08 |
|*  5 |      HASH JOIN             |                   |  971K|   14M|   562   (4)| 00:00:07 |
|*  6 |       HASH JOIN            |                   |    27 |   351 |     5  (20)| 00:00:01 |
|   7 |        TABLE ACCESS FULL   | LOCATIONS         |    23 |   138 |     2   (0)| 00:00:01 |
|   8 |        TABLE ACCESS FULL   | DEPARTMENTS       |    27 |   189 |     2   (0)| 00:00:01 |
|   9 |       INDEX FAST FULL SCAN| EMP_DEPARTMENT_IX |  971K| 2845K|   549   (3)| 00:00:07 |
|* 10 |    INDEX UNIQUE SCAN       | COUNTRY_C_ID_PK   |     1 |    12 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   6 - access("L"."LOCATION_ID"="D"."LOCATION_ID")
  10 - access("C"."COUNTRY_ID"="ITEM_1")
```

Рисунок 5.5 – Вікно з прикладом виконання плану запиту

5.2 Тестування алгоритмів оптимізації плану запитів

5.2.1 Дослідження плану виконання запитів шляхом установки вбудованих режимів оптимізації

Виконуємо запит select * from employees where employee_id = 1000 та отримуємо план виконання запиту відображений у рис. 5.6:

```
-----------------------------------------------------------
| Id  | Operation                    | Name          |
-----------------------------------------------------------
|   0 | SELECT STATEMENT             |               |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |
|*  2 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |
-----------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=1000)
Note
-----
   - rule based optimizer used (consider using cbo)
```

Рисунок 5.6 – План виконання запиту

Аналіз результату: без статистики автоматично використовується RBO-оптимізація; по RBO-оптимізації використовується індексне сканування, використовується UNIQUE – індексу сканування, тому що розглядається операція "=" для колонки типу Primary Key.

Виконуємо запит select * from employees where employee_id < 1000 та отримуємо план виконання запиту відображений на рис. 5.7:

```
-------------------------------------------------------------
| Id  | Operation                    | Name           |
-------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES      |
|*  2 |   INDEX RANGE SCAN           | EMP_EMP_ID_PK |
-------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"<1000)
Note
-----
   - rule based optimizer used (consider using cbo)
```

Рисунок 5.7 – План виконання запиту

Аналіз результату: за RBO-оптимізацією використовується індексне сканування; використовується RANGE індексне сканування, тому що розглядається діапазонна операція "<".

Збираємо загальну статистику за таблицею: analyze table employees compute statistics.

Повторюємо запит select * from emp where empno < 1000 та отримуємо план виконання запиту відображений на рис 5.8:

```
--------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |   900 | 59400 |    14   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |   900 | 59400 |    14   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMP_EMP_ID_PK |   900 |       |     4   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"<1000)
```

Рисунок 5.8 – План виконання запиту

Аналіз результату: при виконанні цього запиту вже не використовується RBO-оптимізація, а CBO-оптимізація. Виконуємо наступний запит select * from employees where department_id = 11 та отримуємо план виконання запиту відображений на рис 5.9:

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows   | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 35967 |  2318K|  8369   (1)| 00:01:41 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES | 35967 |  2318K|  8369   (1)| 00:01:41 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("DEPARTMENT_ID"=11)
```

Рисунок 5.9 – План виконання запиту

Аналіз результату: не використовується індексне сканування, тому що нема індексу по колонці department_id. Індекс автоматично створюється тільки для колонок з обмеження цілісності Primary Key и UNIQUE.

Створюємо індекс типу "бінарне дерево" для колонки department_id таблиці employees: create index emp_department_ix on employees (department_id). Повторюємо запит select * from employees where department_id = 11 та отримуємо план виконання запиту відображений на рис 5.10 :

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows   | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 35967 |  2318K|  8369   (1)| 00:01:41 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES | 35967 |  2318K|  8369   (1)| 00:01:41 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("DEPARTMENT_ID"=11)
```

Рисунок 5.10 – План виконання запиту

Аналіз результату:

- не використовується індексне сканування, тому що фактор селективності по колонці department_id прагне до нуля;
- в колонці department_id нема значень 11, отже повне табличне сканування виконується марно;

– спостерігається помилка CBO-стратегії, т.к. нема статистики за конкретними значеннями колонки, які збираються тільки в гістограмах. Примусова установка режиму RBO-оптимізації:alter session set optimizer_mode = 'rule';

Повторюємо запит select * from employees where department_id = 11 та отримуємо план виконання запиту відображений на рис. 5.11:

```
-----------------------------------------------------------------
| Id  | Operation                    | Name                |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES           |
|*  2 |   INDEX RANGE SCAN           | EMP_DEPARTMENT_IX   |
-----------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"=11)
Note
-----
   - rule based optimizer used (consider using cbo)
```

Рисунок 5.11 – План виконання запиту

Аналіз результату:

– для RBO завжди використовується індексне сканування, якщо у колонці є індекс;

– для значення department_id = 11 RBO-стратегія – обрала кращій план порівняно з CBO-стратегією;

– використовується індексне сканування за алгоритмом index range scan, тому що колонка містить не унікальні значення.

Примусова установка режиму CBO-оптимізації: alter session set optimizer_mode = 'choose';

Збираємо статистику у вигляді гістограми по колонці deptno: analyze table employees compute statistics for columns department_id;

Повторюємо запит select * from employees where department_id = 11 та отримуємо план виконання запиту відображений на рис 5.12:

```
------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |  9264 |  597K|  2483   (1)| 00:00:30 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |  9264 |  597K|  2483   (1)| 00:00:30 |
|*  2 |   INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |  9264 |      |    21   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"=11)
```

Рисунок 5.12 – План виконання запиту

Аналіз результату:

– на основі аналізу гістограми CBO-оптимізація обрано індексне сканування;
– в попередньому прикладі CBO-оптимізація обрала повне сканування, просканував 35967 строк замість 9264 .

Примусова установка режиму RBO-оптимізації: alter session set optimizer_mode = 'rule'.

Виконуємо запит select count(*) from employees where department_id = 11 та отримуємо план виконання запиту відображений на рисунку 5.13:

```
----------------------------------------------------------------
| Id  | Operation            | Name                 |
----------------------------------------------------------------
|   0 | SELECT STATEMENT     |                      |
|   1 |  SORT AGGREGATE      |                      |
|*  2 |   INDEX RANGE SCAN|  EMP_DEPARTMENT_IX    |
----------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"=11)
Note
-----
   - rule based optimizer used (consider using cbo)
```

Рисунок 5.13 – План виконання запиту

Аналіз результату:

– RBO-оптимізації при підрахунку кількості строк таблиці використовує індексне сканування;
– RBO-оптимізація не використовує індекс для виконання операцій проекції (наприклад, сортировки), тому додатково виконується операція table access by index rowid. Примусова установка режиму CBO-оптимизації: alter session set optimizer_mode = 'choose';

Видаляємо статистику за колонками: analyze table employees delete statistics та analyze table employees compute statistics. Повторюємо запит select count(*) from employees where department_id = 11 та отримуємо план виконання запиту відображений на рисунку 5.14:

```
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     1 |     3 |    77   (2)| 00:00:01 |
|   1 |  SORT AGGREGATE    |                   |     1 |     3 |            |          |
|*  2 |   INDEX RANGE SCAN| EMP_DEPARTMENT_IX | 35967 |   105K|    77   (2)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"=11)
```

Рисунок 5.14 – План виконання запиту

Аналіз результату:

– CBO-оптимізація при підрахунку кількості строк таблиці використовує індексне сканування;
– CBO-оптимізація використовує індекс для виконання операцій проекції (наприклад, сортировки), тому додаткових операцій не потрібно, як це було при RBO-оптимізації.

Збираємо статистику у вигляді гістограми по колонці deptno: analyze table employees compute statistics for columns department_id;

Повторюємо запит select count(*) from employees where department_id = 11 та отримуємо план виконання запиту відображений на рис. 5.15:

```
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                     |     1 |     3 |    21   (0)| 00:00:01
|   1 |  SORT AGGREGATE    |                     |     1 |     3 |            |
|*  2 |   INDEX RANGE SCAN| EMP_DEPARTMENT_IX |  9264 | 27792 |    21   (0)| 00:00:01
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"=11)
```

Рисунок 5.15 – План виконання запиту

Аналіз результату: CBO-оптимізація при аналізі гістограми здійснила врахування предиката department_id = 11, просканував 9264 строк замість 35967 строк з попереднього прикладу, та зменшив цей процес більш ніж в 3.5 рази.

5.2.2 Оптимізація та дослідження плану виконання запитів шляхом виправлення неефективної структури запиту

Виконуємо наступний запит та отримуємо план виконання запиту select distinct department_name from departments dep, employees emp where dep.department_id = emp.department_id, відображений на рис. 5.16:

```
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                   |    27 |   513 |   638  (16)| 00:00:08
|   1 |  HASH UNIQUE            |                   |    27 |   513 |   638  (16)| 00:00:08
|*  2 |   HASH JOIN             |                   |  971K|    17M|   560   (4)| 00:00:07
|   3 |    TABLE ACCESS FULL    | DEPARTMENTS       |    27 |   432 |     2   (0)| 00:00:01
|   4 |    INDEX FAST FULL SCAN| EMP_DEPARTMENT_IX |  971K|  2845K|   549   (3)| 00:00:07
------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEP"."DEPARTMENT_ID"="EMP"."DEPARTMENT_ID")
```

Рисунок 5.16 – План виконання запиту

Змінюємо структуру запиту select distinct department_namefrom departments where department_id in (select distinct department_id from employees) за допомогою використання оператора in та отримуємо результат відображений на рисунку 5.17:

```
-------------------------------------------------------------------------------------------
| Id  | Operation              | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                    |    27 |   513 |   638  (16)| 00:00:08 |
|   1 |  HASH UNIQUE           |                    |    27 |   513 |   638  (16)| 00:00:08 |
|*  2 |   HASH JOIN            |                    |  971K|   17M|   560   (4)| 00:00:07 |
|   3 |    TABLE ACCESS FULL   | DEPARTMENTS        |    27 |   432 |     2   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_DEPARTMENT_IX  |  971K|  2845K|   549   (3)| 00:00:07 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DEPARTMENT_ID"="DEPARTMENT_ID")
```

Рисунок 5.17 – План виконання запиту

Аналіз результату: результат показав, що в даному конкретному прикладі нема різниці при використанні операції с in або без in.

Далі змінимо структуру запиту select distinct department_name from departments where exists(select distinct department_id from employees where employees.department_id = departments.department_id) за допомогою оператора exists та отримаємо результат відображений на рисунку 5.18:

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |    27 |   513 |    57   (2)| 00:00:01 |
|   1 |  HASH UNIQUE          |                    |    27 |   513 |    57   (2)| 00:00:01 |
|   2 |   NESTED LOOPS SEMI   |                    |    27 |   513 |    56   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| DEPARTMENTS         |    27 |   432 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN | EMP_DEPARTMENT_IX   |  971K|  2845K|     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("EMPLOYEES"."DEPARTMENT_ID"="DEPARTMENTS"."DEPARTMENT_ID")
```

Рисунок 5.18 – План виконання запиту

Аналіз результату:

– використання операції in дозволило зменшити кількість виконуваних операцій, що значно покращило ефективність виконуємого запиту;
– використання операції in дозволило зменшити кількість скануємих строк в індексі приблизно в 2 рази;
– значно зменшилось число прочитаних байт приблизно в 6 рази.

5.2.3 Примусове використання вбудованих підказок (хинтів)

В наступних прикладах при виконанні запиту будуть використовуватися вбудовані підказки, що, як нам відомо, являють собою коментар, який задається в певному форматі в SQL-команді.

Є декілька категорій підказок та ще більше самих підказок, тому в даної роботі будуть використані тільки деяких з них.

Виконуємо запит select /*+ no_index(employees SYS_C0033223) */ employee_id from employees з примусовим додаванням вбудованої підказки no_index (emp SYS_C0033223) та отримуємо план виконання запиту відображений на рисунку 5.19:

```
--------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   971K|  3793K|  8347   (1)| 00:01:41 |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |   971K|  3793K|  8347   (1)| 00:01:41 |
--------------------------------------------------------------------------------
```

Рисунок 5.19 – План виконання запиту

Виконуємо запит select /*+ index_ffs (employees emp_name_ix) */ first_name from employees з примусовим додаванням вбудованої підказки index_ffs (emp ename_index) та отримуємо план виконання запиту відображений на рисунку 5.20:

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |   971K|  9483K|  4728   (1)| 00:00:57 |
|   1 |  INDEX FAST FULL SCAN| EMP_NAME_IX |   971K|  9483K|  4728   (1)| 00:00:57 |
-------------------------------------------------------------------------------
```

Рисунок 5.20 – План виконання запиту

Видаляємо підказку, додаємо умову та отримуємо план виконання запиту select employee_id from employees where employee_id is not null, відображений на рис. 5.21:

```
---------------------------------------------------------------------------------
| Id  | Operation            | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |               |   971K|  3793K|   564   (3)| 00:00:07 |
|*  1 |  INDEX FAST FULL SCAN| EMP_EMP_ID_PK |   971K|  3793K|   564   (3)| 00:00:07 |
---------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("EMPLOYEE_ID" IS NOT NULL)
```

Рисунок 5.21 – План виконання запиту

Аналіз результату: підказка no_index (emp SYS_C0033223) примусово вимкнула використання системного індексу та призвела до використання найменш ефективного доступу до даних, а саме table access full, що в свою чергу призвело до більшого витрачання усіх ресурсів наданих в плані виконання. Підказка index_ffs (employees emp_name_ix) зробила план виконання більш ефективним, за допомогою примусового використання створеного індексу emp_name_ix витрачений час на виконання зменшився з 101 секунд до 57, що при великої кількості записів може значно покращити роботу системи. Самий кращий результат показав запит з перевіркою на наявність значень, який зменшив запит до 7 секунд. Також потрібно зазначити, що виконання запиту в звичайному режимі призвело до значної оптимізації плану виконання запиту. Виконуємо запит select /*+star */ c.country_name, (COUNT(e.last_name)) as t from countries c,employees e,departments d,locations l where c.country_id = l.country_id and l.location_id = d.location_id and d.department_id = e.department_id group by c.country_name з примусовим додаванням вбудованої підказки /*+ star */ , що визначає порядок доступу до таблиць, та отримуємо план виконання запиту відображений на рисунку 5.22:

```
---------------------------------------------------------------------------------------------
| Id  | Operation                | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                     |    25 |   700 |   641  (16)| 00:00:08 |
|   1 |  HASH GROUP BY           |                     |    25 |   700 |   641  (16)| 00:00:08 |
|*  2 |   HASH JOIN              |                     |  971K|   25M|   562   (4)| 00:00:07 |
|*  3 |    HASH JOIN             |                     |    27 |   675 |     5  (20)| 00:00:01 |
|   4 |     NESTED LOOPS         |                     |    23 |   414 |     2   (0)| 00:00:01 |
|   5 |      TABLE ACCESS FULL   | LOCATIONS           |    23 |   138 |     2   (0)| 00:00:01 |
|*  6 |      INDEX UNIQUE SCAN   | COUNTRY_C_ID_PK     |     1 |    12 |     0   (0)| 00:00:01 |
|   7 |     TABLE ACCESS FULL    | DEPARTMENTS         |    27 |   189 |     2   (0)| 00:00:01 |
|   8 |    INDEX FAST FULL SCAN| EMP_DEPARTMENT_IX   |  971K|  2845K|   549   (3)| 00:00:07 |
---------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   3 - access("L"."LOCATION_ID"="D"."LOCATION_ID")
   6 - access("C"."COUNTRY_ID"="L"."COUNTRY_ID")
```

Рисунок 5.22 – План виконання запиту

Виконуємо запит select /*+ordered*/c.country_name, (COUNT (e.last_name)) as t from countries c,employees e,departments d,locations lwhere c.country_id = l.country_id and l.location_id = d.location_id and d.department_id = e.department_id group by c.country_name з примусовим додаванням вбудованої підказки /*+ordered*/ та отримуємо план виконання запиту відображений на рисунку 5.23:

```
----------------------------------------------------------------------------------------------
| Id  | Operation                  | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                   |    14 |   392 |   642  (16)| 00:00:08 |
|   1 |  HASH GROUP BY             |                   |    14 |   392 |   642  (16)| 00:00:08 |
|   2 |   NESTED LOOPS             |                   |    14 |   392 |   641  (16)| 00:00:08 |
|   3 |    VIEW                    | VW_GBF_25         |    14 |   224 |   641  (16)| 00:00:08 |
|   4 |     HASH GROUP BY          |                   |    14 |   224 |   641  (16)| 00:00:08 |
|*  5 |      HASH JOIN             |                   |  971K|   14M|   562   (4)| 00:00:07 |
|*  6 |       HASH JOIN            |                   |    27 |   351 |     5  (20)| 00:00:01 |
|   7 |        TABLE ACCESS FULL   | LOCATIONS         |    23 |   138 |     2   (0)| 00:00:01 |
|   8 |        TABLE ACCESS FULL   | DEPARTMENTS       |    27 |   189 |     2   (0)| 00:00:01 |
|   9 |       INDEX FAST FULL SCAN| EMP_DEPARTMENT_IX |  971K|  2845K|   549   (3)| 00:00:07 |
|* 10 |    INDEX UNIQUE SCAN       | COUNTRY_C_ID_PK   |     1 |    12 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   6 - access("L"."LOCATION_ID"="D"."LOCATION_ID")
  10 - access("C"."COUNTRY_ID"="ITEM_1")
```

Рисунок 5.23 – План виконання запиту

Аналіз результату: на основі отриманих планів виконання запитів можна зробити висновок, що при використанні /*+star */ вбудований оптимізатор обирає кращий, більш ефективний план виконання запиту, він обходить стороною порядок переліку таблиць в запиті, та формує власний порядок обробки таблиць, що призводить до зменшення використання ресурсів на виконання запиту порівняно з підказкою /*+ordered */, що примушує систему виконувати запит з обробкою таблиць саме в тому порядку, в якому вони є у за-

питі та призводить до збільшення кількості операцій у плані виконання та ресурсів витрачених на виконання.

ВИСНОВКИ

Магістерська робота присвячена вирішенню актуальної проблеми оптимізації обробки даних. Необхідність збільшення швидкості аналізу даних, що зумовлена реальністю предметних областей, має на увазі не тільки збільшення обчислювальних потужностей, але і опрацювання на інших рівнях.

В результаті виконання магістерської роботи проведено дослідження існуючих алгоритмів оптимізації планів виконання запитів у системі управління базами даних Oracle.

Проведено аналіз і оцінка різноманітних шляхів оптимізації планів виконання запитів, перелік яких постійно поповнюється новими сучасними елементами. Як зазначено у другому розділі роботи, не дивлячись на різноманіття алгоритмів оптимізації, проблема обробки даних не зникла. Дані можуть бути будь-якого формату, укладені у будь-яку кількість таблиць, можуть мати складну ієрархічну структуру або мати тісний зв’язок між собою, їх об’єм може зростати з геометричною прогресією, тому на сьогоднішній день не винайдено ідеального або універсального алгоритму обробки даних.

Здійснена програмна реалізація підсистеми дослідження аналізу алгоритмів оптимізації виконання SQL-запитів за допомогою якої можна досліджувати та аналізувати алгоритми оптимізації плану виконання запитів.

За допомогою підсистеми, користувач буде бачити, які ресурси і в яких об’ємах були витрачені, та обирати найбільш відповідний до його вимог запит. Виконана наочна демонстрація результатів оптимізації планів виконання SQL-запитів, засобами розробленої підсистеми.

ПЕРЕЛІК ПОСИЛАНЬ

1. Алан Бьюли. Изучаем SQL. – М.: Издательство: Символ-Плюс, 2007. – 312 с.
2. Д. Кренке. Теория и практика построения баз данных. – СПб.: Питер, 2005. – 864 с.
3. Завадський І.О. Основи баз даних. – К.: Видавець, 2011. – 192 с.
4. Томас Кайт, Кун Дарл. Oracle для профессионалов. – М.: Вильямс, 2016. – 960 с.
5. Грофф Дж. Р., Вайнберг П.Н., Оппелъ Э. Дж. SQL. Полное руководство. – М.: Вильямс, 2015. – 959 с.
6. Гринвальд Р., Стерн Дж. Oracle 11g. – М.: Символ-Плюс, 2009. – 464 с.
7. Л. В. Рудикова. Базы данных. Разработка приложений. – БХВ-Петербург, 2006. – 469 с.
8. Джонатан Льюис. Oracle. Основы стоимостной оптимизации. – СПб.: Питер, 2007. – 528 с.
9. Герберт Шилдт. Swing. Руководство для начинающих. – М.: Вильямс, 2007. – 704 с.
10. Джез Хамбл, Дейвид Фарли. Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ. – М.: Вильямс, 2016. – 432 с.
11. Джошуа Блох. Java. Эффективное программирование. – М.: Лори, 2014. – 310 с.
12. О Java [Електроний ресурс] – Режим доступу: http://fulcrum81.gitbooks.io/trial–book/content/java_intro /README.html
13. Крис Дж. Дейт. Введення в системи баз даних. Пер. з англ. – К.: Птицин, 2006. – 1328 с.
14. Карпова Т.С. Базы данных: модели, разработка, реализация. – СПб.: Питер, 2001. – 304 с.

ДОДАТОК А Основні коди програмних модулів

```
package db_client;
import javax.swing.*;
public class NewConnectionPanel extends javax.swing.JPanel {
    public void setConnectionString()
    {
        JdbcThin.connectURL ="jdbc:oracle:thin:" +
this.jTextFieldUserName.getText().trim() +
            "/" +
            this.jTextFieldPassword.getText().trim() +
            "@" +
            this.jTextFieldHost.getText().trim() +
            ":" +
            this.jTextFieldPort.getText().trim() +
            ":" +
            this.jTextFieldDB.getText().trim() ;
    }
        /** Creates new form NewConnectionPanel */
    public NewConnectionPanel() {
        initComponents();
    }
    @SuppressWarnings("unchecked")
    private void initComponents() {
        jLabelUserName = new JLabel();
        jLabelpassword = new JLabel();
        jLabelHost = new JLabel();
        jLabelPort = new JLabel();
        jLabelInstanceName = new JLabel();
        jTextFieldUserName = new JTextField();
        jTextFieldPassword = new JTextField();
        jTextFieldHost = new JTextField();
        jTextFieldPort = new JTextField();
        jTextFieldDB = new JTextField();
        setBorder(BorderFactory.createTitledBorder("New Connection"));
        jLabelUserName.setText("Username");
        jLabelpassword.setText("Password");
        jLabelHost.setText("Host");
        jLabelPort.setText("Port");
        jLabelInstanceName.setText("Instance Name");
        jTextFieldHost.setText("localhost");
        jTextFieldPort.setText("1521");
        jTextFieldDB.setText("ORCL");
        GroupLayout layout = new GroupLayout(this);
        this.setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
```

```
                    .addGap(48, 48, 48)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
                         .addComponent(jLabelpassword)
                         .addComponent(jLabelUserName)
                         .addComponent(jLabelHost)
                         .addComponent(jLabelPort)
                         .addComponent(jLabelInstanceName))

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING,
false)

.addComponent(jTextFieldUserName,GroupLayout.DEFAULT_SIZE, 128,
Short.MAX_VALUE)
                         .addComponent(jTextFieldPassword)
                         .addComponent(jTextFieldHost)
                         .addComponent(jTextFieldPort)
                         .addComponent(jTextFieldDB))
                    .addContainerGap())
          );
          layout.setVerticalGroup(
               layout.createParallelGroup(GroupLayout.Alignment.LEADING)
               .addGroup(layout.createSequentialGroup()
                    .addContainerGap()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                         .addComponent(jTextFieldUserName,
GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                         .addComponent(jLabelUserName))

.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                         .addComponent(jTextFieldPassword,
GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                         .addComponent(jLabelpassword))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                         .addComponent(jTextFieldHost,
GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                         .addComponent(jLabelHost))
```

```
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                    .addComponent(jTextFieldPort,
GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                    .addComponent(jLabelPort))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                    .addComponent(jTextFieldDB,
GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                    .addComponent(jLabelInstanceName))
                .addContainerGap(GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );
    }
    private JLabel jLabelUserName;
    private JLabel jLabelpassword;
    private JLabel jLabelHost;
    private JLabel jLabelPort;
    private JLabel jLabelInstanceName;
    private JTextField jTextFieldUserName;
    private JTextField jTextFieldPassword;
    private JTextField jTextFieldHost;
    private JTextField jTextFieldPort;
    private JTextField jTextFieldDB;
}

package db_client;
import java.sql.*;
import javax.swing.JOptionPane;
public class JdbcThin {
        static String connectURL;
        static java.sql.Connection connection;
    java.sql.Statement statement;
    java.sql.ResultSet resultSet;
    static oracle.jdbc.pool.OracleDataSource oracleDataSource;
        Object data[][];
    Object column[];
           void fillArrays(String query){
       int i=0,j=0;
       int countRow;
       int countColumn;
             data = new Object[0][0];
```

```
        column = new Object[0];
               try
        {
            statement =
connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                          ResultSet.CONCUR_READ_ONLY);
            resultSet = statement.executeQuery(query);
            ResultSetMetaData rsMetaData = resultSet.getMetaData();
            countColumn = rsMetaData.getColumnCount();
            resultSet.last();
            countRow = resultSet.getRow();
            data = (Object[][]) JdbcThin.expand(data, countRow);
            i=0;
            while (i< data.length){
                if (data[i] == null)
                  data[i] = new Object[countColumn];
                else
                  data[i] = (Object[]) JdbcThin.expand(data[i],
countColumn);
              i++;
            }
            column = new Object[countColumn];
            resultSet.first();
            i=0;
            while(i<countRow){
                j=0;
                while(j<countColumn){
                  data[i][j]= resultSet.getString(j+1);
                  j++;
                }
                i++;
                resultSet.next();
            }
            i=0;
            while (i<countColumn){
            column[i] = rsMetaData.getColumnName(i+1);
            i++;
            }
        }
        catch (Exception ex)
        {
            JOptionPane.showMessageDialog(DBClient.myFrame , ex,
"Error", JOptionPane.ERROR_MESSAGE);
        }

    }
      static void connectDB()
    {
```

```
        try
        {
            JOptionPane.showMessageDialog(DBClient.myFrame , new
Exception("start"), "Error", JOptionPane.ERROR_MESSAGE);
            oracleDataSource = new oracle.jdbc.pool.OracleDataSource();

            JOptionPane.showMessageDialog(DBClient.myFrame , new
Exception("get oracle"), "Error", JOptionPane.ERROR_MESSAGE);

            oracleDataSource.setURL(connectURL);

            JOptionPane.showMessageDialog(DBClient.myFrame , new
Exception("set conenction" + oracleDataSource + " "), "Error",
JOptionPane.ERROR_MESSAGE);
            JOptionPane.showMessageDialog(DBClient.myFrame , new
Exception("set conenction" + oracleDataSource.getServerName()),
"Error", JOptionPane.ERROR_MESSAGE);
            connection = oracleDataSource.getConnection();

            JOptionPane.showMessageDialog(DBClient.myFrame , new
Exception("Connected"), "Error", JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception ex)
        {

        JOptionPane.showMessageDialog(DBClient.myFrame , ex, "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }
      void disconnectDB() throws SQLException {
            if (!(connection.isClosed())) {
              connection.close();
            }
            oracleDataSource.close();
    }

     private static Object expand(Object a, int size) {

         //if the object class is not an array then exit the function
         Class cl = a.getClass();
         if (!cl.isArray()) return null;

         //get the size of the array
         int length = java.lang.reflect.Array.getLength(a);
         //int newLength = length + (length / 2); // 50% more
         int newLength = size; // 50% more

         //resize the array and return the new array
```

```
        Class componentType = a.getClass().getComponentType();
        Object newArray =
java.lang.reflect.Array.newInstance(componentType, newLength);
        System.arraycopy(a, 0, newArray, 0, length);
        return newArray;
    }
}
package db_client;

import jsyntaxpane.DefaultSyntaxKit;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;
import javax.swing.table.TableModel;
import java.awt.event.KeyEvent;

public class frameMain extends JFrame {

    private  JDialog newConnDialog =  new
dbClient.dialogNewConnection(this ,true);

    public frameMain() {
        initComponents();
        jSplitPane1.setDividerLocation(150);
        jSplitPane1.setDividerSize(3);

        DefaultSyntaxKit.initKit();
        jEditorPane1.setContentType("text/sql");

        String aData[][] = new String[0][0];
        String aCol[] = new String[0];

         TableModel tModel =
                new  DefaultTableModel(aData,aCol);
        jTable1.setModel(tModel);

    }
    @SuppressWarnings("unchecked")
    private void initComponents() {
        jSplitPane1 = new  JSplitPane();
        jScrollPane2 = new  JScrollPane();
        jPanel1 = new  JPanel();
        jInternalFrame1 = new  JInternalFrame();
        jScrollPane3 = new  JScrollPane();
        jTable1 = new  JTable();
        jScrollPane1 = new  JScrollPane();
        jEditorPane1 = new  JEditorPane();
        jMenuBar1 = new  JMenuBar();
```

```
        jMenu1 = new  JMenu();
        jMenuItem2 = new  JMenuItem();
        jMenuItem6 = new  JMenuItem();
        jMenu2 = new  JMenu();
        jMenu3 = new  JMenu();
        jMenuItem1 = new  JMenuItem();
        jSeparator1 = new  JPopupMenu.Separator();
        jMenuItem3 = new  JMenuItem();
        jMenuItem4 = new  JMenuItem();
        jSeparator2 = new  JPopupMenu.Separator();
        jMenuItem5 = new  JMenuItem();
        setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE);
        jSplitPane1.setOrientation( JSplitPane.VERTICAL_SPLIT);
        jPanel1.setMaximumSize(new java.awt.Dimension(327670,
327670));
        jPanel1.setRequestFocusEnabled(false);
        jInternalFrame1.setMaximizable(true);
        jInternalFrame1.setTitle("Result");
        jInternalFrame1.setAutoscrolls(true);
        jInternalFrame1.setFont(new java.awt.Font("Courier New", 0,
10));
        jInternalFrame1.setPreferredSize(new java.awt.Dimension(1200,
1200));
        jInternalFrame1.setRequestFocusEnabled(false);
        jInternalFrame1.setVisible(true);

        jScrollPane3.setPreferredSize(getPreferredSize());

        jTable1.setModel(new   DefaultTableModel(
            new Object [][] {
                {null, null, null, null},
                {null, null, null, null},
                {null, null, null, null},
                {null, null, null, null}
            },
            new String [] {
                "Title 1", "Title 2", "Title 3", "Title 4"
            }
        ));
        jTable1.setAutoscrolls(false);
        jScrollPane3.setViewportView(jTable1);
         GroupLayout jInternalFrame1Layout = new
GroupLayout(jInternalFrame1.getContentPane());

jInternalFrame1.getContentPane().setLayout(jInternalFrame1Layout);
        jInternalFrame1Layout.setHorizontalGroup(
            jInternalFrame1Layout.createParallelGroup(
GroupLayout.Alignment.LEADING)
```

```
            .addComponent(jScrollPane3,  GroupLayout.DEFAULT_SIZE,
1190, Short.MAX_VALUE)
        );
        jInternalFrame1Layout.setVerticalGroup(
            jInternalFrame1Layout.createParallelGroup(
GroupLayout.Alignment.LEADING)
            .addComponent(jScrollPane3,  GroupLayout.DEFAULT_SIZE,
1168, Short.MAX_VALUE)
        );
         GroupLayout jPanel1Layout = new  GroupLayout(jPanel1);
        jPanel1.setLayout(jPanel1Layout);
        jPanel1Layout.setHorizontalGroup(
            jPanel1Layout.createParallelGroup(
GroupLayout.Alignment.LEADING)
            .addGroup(jPanel1Layout.createSequentialGroup()
                .addComponent(jInternalFrame1,
GroupLayout.PREFERRED_SIZE,  GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                .addContainerGap(29, Short.MAX_VALUE))
        );
        jPanel1Layout.setVerticalGroup(
            jPanel1Layout.createParallelGroup(
GroupLayout.Alignment.LEADING)
            .addGroup(jPanel1Layout.createSequentialGroup()
                .addComponent(jInternalFrame1,
GroupLayout.PREFERRED_SIZE,  GroupLayout.DEFAULT_SIZE,
GroupLayout.PREFERRED_SIZE)
                .addContainerGap(317, Short.MAX_VALUE))
        );
        jScrollPane2.setViewportView(jPanel1);
        jSplitPane1.setRightComponent(jScrollPane2);
        jEditorPane1.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(java.awt.event.KeyEvent evt) {
                jEditorPane1KeyPressed(evt);
            }
        });
        jScrollPane1.setViewportView(jEditorPane1);
        jSplitPane1.setLeftComponent(jScrollPane1);
        jMenuItem2.setText("New Connection");
        jMenuItem2.addMouseListener(new java.awt.event.MouseAdapter()
{
            public void mousePressed(java.awt.event.MouseEvent evt) {
                jMenuItem2MousePressed(evt);
            }
        });
        jMenu1.add(jMenuItem2);
        jMenuBar1.add(jMenu1);
```

```
        jMenu2.setText("Edit");
        jMenuBar1.add(jMenu2);
        jMenu3.setText("Run");
        jMenuItem1.setText("Run SQL (F5)");
        jMenuItem1.addMouseListener(new java.awt.event.MouseAdapter()
{
            public void mousePressed(java.awt.event.MouseEvent evt) {
                jMenuSQLF5MousePressed(evt);
            }
        });
        jMenu3.add(jMenuItem1);
        jMenu3.add(jSeparator1);
        jMenuBar1.add(jMenu3);
        setJMenuBar(jMenuBar1);
         GroupLayout layout = new  GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup( GroupLayout.Alignment.LEADING)
            .addComponent(jSplitPane1,  GroupLayout.DEFAULT_SIZE, 650,
Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup( GroupLayout.Alignment.LEADING)
            .addComponent(jSplitPane1,  GroupLayout.DEFAULT_SIZE, 602,
Short.MAX_VALUE)
        );
        pack();
    }
    public static void showFrame()
    {
        frameMain.jSplitPane1.setVisible(true);
    }

/*
 * run SQL(F5) menu item
 */
private void jMenuSQLF5MousePressed(java.awt.event.MouseEvent evt) {
    this.myData = new JdbcThin();
    myData.connectDB();
    myData.fillArrays(this.jEditorPane1.getText().trim());
      TableModel tModel = new   DefaultTableModel(myData.data,
myData.column);
    this.jTable1.setModel(tModel);
      TableColumn column = null;
    int colcount = this.jTable1.getColumnCount();
    int rowcount = this.jTable1.getRowCount();
    int totalwidth = 0;
    for (int i = 0; i < colcount; i++) {
```

```
        column = jTable1.getColumnModel().getColumn(i);
        column.setPreferredWidth(myData.column[i].toString().length()
* 25);
        totalwidth = totalwidth +
(myData.column[i].toString().length() * 20);
    }
    if (rowcount != 1) {
        jInternalFrame1.setSize(totalwidth, rowcount * 20);
        jInternalFrame1.setPreferredSize(new
java.awt.Dimension(totalwidth, rowcount * 20));
    } else {
        jInternalFrame1.setSize(totalwidth, 200);
        jInternalFrame1.setPreferredSize(new
java.awt.Dimension(totalwidth, 200));
    }
    jInternalFrame1.repaint();
}
/*
 * New Connection Menu Item
 */
private void jMenuItem2MousePressed(java.awt.event.MouseEvent evt) {

    this.newConnDialog.setLocationRelativeTo(this.jSplitPane1);
    this.newConnDialog.setTitle("New Connection");
    this.newConnDialog.setVisible(true);
   }
private void jEditorPane1KeyPressed(KeyEvent evt) {
   String tts="";
   int tt=0;
   tt = evt.getKeyCode();
   if (tt==116) {
    this.myData = new JdbcThin();
      myData.connectDB();
    myData.fillArrays(this.jEditorPane1.getText().trim());
    myData.disconnectDB();
       TableModel tModel = new
DefaultTableModel(myData.data,myData.column);
    this.jTable1.setModel(tModel);
    TableColumn column = null;
    int colcount = this.jTable1.getColumnCount();
    int rowcount = this.jTable1.getRowCount();
    int totalwidth=0;
        for (int i = 0; i < colcount; i++) {
      column = jTable1.getColumnModel().getColumn(i);
column.setPreferredWidth(myData.column[i].toString().length()*25);
      totalwidth =
totalwidth+(myData.column[i].toString().length()*20);
    }
```

```
    //adjust the size of the internal frame
    //accordong to the size of the data filled in the jTable
    if (rowcount!=1)
    {
      jInternalFrame1.setSize(totalwidth, rowcount*20);
      jInternalFrame1.setPreferredSize(new
java.awt.Dimension(totalwidth,rowcount*20));
    }
    else
    {
      jInternalFrame1.setSize(totalwidth, 200);
      jInternalFrame1.setPreferredSize(new
java.awt.Dimension(totalwidth,200));
    }
    jInternalFrame1.repaint();
   }
}
private void jMenuItem4MousePressed(java.awt.event.MouseEvent evt) {
    FileOS myOps = new FileOS();
     JFileChooser fc = new  JFileChooser();
    fc.showSaveDialog(this);
    myOps.write(this.jEditorPane1.getText() ,
fc.getSelectedFile().toString());
}
private void jMenuItem3MousePressed(java.awt.event.MouseEvent evt) {
    FileOS myOps = new FileOS();
     JFileChooser fc = new  JFileChooser();
    fc.showOpenDialog(this);

this.jEditorPane1.setText(myOps.read(fc.getSelectedFile().toString()))
;
}
    public JdbcThin myData;
    public  JEditorPane jEditorPane1;
    public static  JInternalFrame jInternalFrame1;
    public static  JMenu jMenu1;
    public static  JMenu jMenu2;
    public static  JMenu jMenu3;
    public static  JMenuBar jMenuBar1;
    public static  JMenuItem jMenuItem1;
    public static  JMenuItem jMenuItem2;
    public static  JMenuItem jMenuItem3;
    public static  JMenuItem jMenuItem4;
    public static  JMenuItem jMenuItem5;
    public static  JMenuItem jMenuItem6;
    public static  JPanel jPanel1;
    public static  JScrollPane jScrollPane1;
    public static  JScrollPane jScrollPane2;
```

```
    public static  JScrollPane jScrollPane3;
    public static  JPopupMenu.Separator jSeparator1;
    public static  JPopupMenu.Separator jSeparator2;
    public static  JSplitPane jSplitPane1;
    public static  JTable jTable1;
}
package db_client;
import java.io.*;
class FileOS {
    void write(String str, String name)
    {
    FileOutputStream outputStream;
    PrintStream printStream;
    try
    {
        outputStream = new FileOutputStream(name);
        printStream = new PrintStream(outputStream);
        printStream.println(str);
        printStream.close();
    }
    catch (Exception ex)
    {
        javax.swing.JOptionPane.showMessageDialog(DBClient.myFrame,
                "classsFileOps.write: \n" + ex,
                "Error",
                javax.swing.JOptionPane.ERROR_MESSAGE);
    }
    }
    String read(String name)
    {
    String file="";
    java.io.BufferedReader bufferedReader ;
    try
    {
        bufferedReader = new java.io.BufferedReader(new
java.io.FileReader(name));
        while (true)
        {
            String str = bufferedReader.readLine();
            if (str == null)
            {
              break;
            }
            else
            {
              file += str + "\n";
            }
        }
```

```
    }
    catch (Exception ex)
    {
        javax.swing.JOptionPane.showMessageDialog(DBClient.myFrame,
                "classsFileOps.read: \n" + ex,
                "Error",
                javax.swing.JOptionPane.ERROR_MESSAGE);
    }
    return (file);
    }
}
package db_client;
public class dialogNewConnection extends javax.swing.JDialog {
    /** Creates new form dialogNewConnection */
    public dialogNewConnection(java.awt.Frame parent, boolean modal) {
        super(parent, modal);
        initComponents();
    }
    @SuppressWarnings("unchecked")
    private void initComponents() {
        newConnectionPanel1 = new NewConnectionPanel();
        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();
setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE)
;
        jButton1.setText("Connect");
        jButton1.addActionListener(new java.awt.event.ActionListener()
{
         public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        jButton2.setText("Cancel");
        jButton2.addActionListener(new java.awt.event.ActionListener()
{
            public void actionPerformed(java.awt.event.ActionEvent
evt) {
                jButton2ActionPerformed(evt);
            }
        });
        javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
```

```
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment
.TRAILING)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jButton2)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(jButton1))
                    .addComponent(newConnectionPanel1,
javax.swing.GroupLayout.PREFERRED_SIZE, 291,
javax.swing.GroupLayout.PREFERRED_SIZE))
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );
        layout.setVerticalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addComponent(newConnectionPanel1,
javax.swing.GroupLayout.PREFERRED_SIZE, 190,
javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment
.BASELINE)
                    .addComponent(jButton1)
                    .addComponent(jButton2))
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
        );
        pack();
    }
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    this.newConnectionPanel1.setConnectionString();
    this.setVisible(false);
}
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    this.setVisible(false);
}
    public static javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private NewConnectionPanel newConnectionPanel1;
}
package db_client;
import javax.swing.*;
public class DBClient {
    public static JFrame myFrame = new frameMain();
    public static void main(String[] args)
```

```
        myFrame.setSize(800, 600);
        myFrame.setVisible(true);
    }
}
```