

## Тема лекції 10:

# Оптимізація запитів до бази даних

---

- ❑ Використання індексів в системах з базами даних
  - ❑ Рекомендації з використання індексів в SQL Server
  - ❑ Використання кластерних індексів
  - ❑ Використання некластерних індексів
  - ❑ Рекомендації щодо написання запитів на вибірку даних
-

# Використання індексів в системах з базами даних

---

- ❑ З погляду користувача, індекс – це перелік стовпців таблиці, за значеннями яких записи логічно впорядковуються.
  - ❑ З погляду СУБД, індекс – це механізм, що дає змогу значно підвищити швидкість доступу до записів за індексованими полями та забезпечує ефективну перевірку унікальності значень індексованих полів.
-

# Індекси забезпечують:

---

- ❑ задоволення вимоги унікальності записів
  - ❑ підтримку логічної упорядкованості даних відповідно до значень одного чи декількох полів
  - ❑ оптимізацію виконання запитів
-

# Правила використання індексів

---

- ❑ у таблицях невеликих розмірів індекси майже не забезпечують підвищення продуктивності
  - ❑ продуктивність значно підвищується в тих випадках, коли стовпці містять переважно неповторювані дані чи багато NULL-значень
  - ❑ завдяки індексам оптимізується виконання запитів, що видають невелику кількість результатних рядків (до 25%)
  - ❑ **слід пам'ятати**, що індекси прискорюють пошук даних, однак сповільнюють процес їхнього оновлення, що стає особливо відчутним під час одночасного оновлення великої кількості рядків. (У подібних випадках перед оновленням індекс потрібно видаляти, а після завершення даної операції – відновити)
-

# Правила використання індексів

---

- ❑ зберігання індексів потребує значних обсягів пам'яті. Якщо СУБД дає змогу керувати пам'яттю, слід відвести частину пам'яті під індекси;
  - ❑ потрібно завжди індексувати поля, що використовуються для з'єднання таблиць – це значно прискорює виконання запитів;
  - ❑ не слід індексувати поля, які регулярно оновлюються;
  - ❑ не бажано зберігати індекси разом з таблицями на одному фізичному пристрої. Розподіл цих об'єктів між носіями інформації знижує навантаження на них та прискорює виконання запитів.
-

# Прості та складені індекси

---

- ❑ Індекси можуть бути побудовані на декількох полях.
  - ❑ Якщо для одного індексу вказується більш ніж одне поле, то друге поле впорядковується всередині першого, третє всередині другого і т.д. Це виконується незалежно від способу впорядкування стовпців у таблиці.
  - ❑ У складених індексах слід спочатку зазначати поля, що використовуються найчастіше.
  - ❑ Складені індекси потрібно застосовувати тоді, коли зазначені в них поля використовуються для опису умови вибору даних.
-

# Рекомендації з використання індексів в SQL Server

---

- ❑ Основним засобом підвищення швидкості виконання запитів на вибірку даних є створення необхідних індексів.
  - ❑ Перед тим як створювати оптимальні індекси для таблиць необхідно **передбачити, які запити** і в які моменти часу будуть виконуватись в базі даних, що є непростю задачею, особливо, якщо робляться спроби додати індекси до нової бази даних.
  - ❑ Оптимізуючи індекси для бази даних, насамперед необхідно **ідентифікувати проблемні запити**. Це запити, які найчастіше виконуються, а також такі, які є найбільш трудомісними з точки зору затрат ресурсів, які виконуються у моменти загального навантаження на сервер.
-

# Типи індексів для оптимізації виконання запитів

---

- ☐ кластерний або некластерний
  - ☐ унікальний або неунікальний
  - ☐ простий чи складений
  - ☐ повнотабличні або фільтровані некластерні індекси
-



# Параметри індексів для оптимізації виконання запитів

---

- ❑ Можна задати початкові характеристики сховища індексу, щоб оптимізувати його продуктивність або підтримку, задавши такий його параметр як FILLFACTOR.
  - ❑ Додатково можна визначити місце зберігання індексу за допомогою файлових груп або схем секціонування.
-

# Використання кластерних індексів

---

- ❑ Кластерний індекс відповідає фізичному порядку сортування даних в таблиці. Звідси випливає, що таблиця може мати лише один кластерний індекс.
  - ❑ Як правило, цей кластерний індекс використовують для реалізації обмеження первинного ключа. Тому кластерний індекс володіє обмеженням унікальності.
  - ❑ Умова унікальності не є обов'язковою. Однак, неунікальний кластерний індекс автоматично включає в себе додаткову інформацію, розміром 4байта, на кожен індексний запис для того, щоб гарантувати їх унікальність на скритому від користувача рівні. Це призводить до збільшення розміру кластерного індексу.
-

# Правило 1 кластеризації таблиць

---

- ***Кожна таблиця повинна мати кластерний індекс***
  - Пояснення цьому правилу пов'язане з особливістю SQL Server зчитувати дані.
    - Коли зчитуються дані, які зберігаються в кластерному індексі, SQL Server має можливість за одну елементарну операцію читати цілий екстент (8 послідовних сторінок або 64 Кбайта).
    - Для порівняння – некластерні індекси і дані з таблиць зчитуються посторінково (8 Кбайт за одну елементарну операцію читання). Тому перевага від наявності кластерних індексів є очевидною.
  - Наявність кластерного індексу в таблиці гарантує, що сторінки даних цієї таблиці будуть дефрагментовані при операції перебудови індексів.
-

# Правило 2 кластеризації таблиць

---

- ***Кластерний індекс треба створювати перед створенням будь-яких некластерних індексів.***
  - Якщо такий порядок порушити, то при створенні кластерного індексу усі некластерні індекси будуть перебудовані автоматично, що є довготривалим процесом.
-

# Правило 3 кластеризації таблиць

---

- ***Якщо таблиця має і кластерний і некластерні індекси, то рекомендується, щоб кластерний індекс базувався на єдиному стовпці мінімальної довжини.***
  - Часто цю умову задовольняє атомарний автоінкрементний первинний ключ таблиці.
    - У цьому випадку є ще одна перевага, а саме, виключається розщеплення сторінок даних. Тобто якщо в таблицю з автоінкрементним кластерним індексом вставляються нові рядки, то ці вставки фізично реалізуються послідовно одна за другою, і не буде відбуватись розбиття сторінок.
  - У загальному випадку необхідно так визначити кластерний індекс, щоб в нього увійшло якомога **менше** стовпців.
-

# Правило 4 кластеризації таблиць

---

- Неефективним є вибір широких атрибутів для кластерних індексів.
    - Ключові значення з кластерного індексу використовуються всіма некластерними індексами як уточнюючі. Тому будь-які некластерні індекси, визначені на тій же таблиці, будуть значно більшими.
-

# Атрибути – претенденти на кластерний індекс

---

- унікальні атрибути або ті, які містять багато різних значень
  - звернення до цих атрибутів відбувається послідовно
  - атрибути, які часто використовуються для сортування даних
-

# Випадки застосування кластерних індексів

---

- Запити повертають діапазон значень за допомогою предикатів BETWEEN, >, >=, < і <=.
- У таких випадках після того як знайдено перший рядок за допомогою кластерного індексу, рядки з наступними індексованими значеннями гарантовано будуть фізично суміжними зі знайденим рядком.
- Запити повертають великі результатні набори.
- Запити містять операцію з'єднання таблиць (JOIN);
  - як правило, в них беруть участь і стовпці зовнішнього ключа.



# Випадки застосування кластерних індексів

---

- ❑ Запити використовують фрази GROUP BY або ORDER BY.
    - У цих випадках індекс за стовпцями, вказаними в таких фразах, виключає потребу у сортуванні даних, оскільки рядки вже посортовані. Це й покращує продуктивність запиту.
  - ❑ Запити з групуванням і агрегатними функціями, такими як MAX, MIN або COUNT, оскільки усі ці операції виконуються через проміжну операцію сортування даних.
-

# Використання некластерних індексів

---

- ❑ Некластерні індекси створюють для підвищення ефективності запитів, які часто використовуються і не враховуються кластерними індексами.
  - ❑ При проектуванні некластерних індексів необхідно враховувати характеристики систем з базами даних.
-

# Використання некластерних індексів в OLAP-системах

---

- ❑ Запити до бази даних можуть виконуватись швидше при використанні великої кількості некластерних індексів.
  - ❑ У процесі визначення найшвидшого метода доступу оптимізатор запитів може вибирати з великої кількості індексів.
  - ❑ Якщо база даних не призначена для частих оновлень, то операції з обслуговування індексів не будуть впливати на швидкодію системи.
-

# Використання некластерних індексів в OLTP-системах

---

- ❑ Необхідно уникати великої кількості некластерних індексів.
  - ❑ Крім того, індекси повинні бути вузькими, тобто містити мінімальну кількість стовпців.
  - ❑ Велика кількість індексів в таблиці знижує швидкість виконання операцій INSERT, UPDATE та DELETE, оскільки при зміні даних усі індекси повинні перебудуватись.
-

# Поняття селективності стовпця

---

## □ Селективність стовпця $S$

називається відношення кардинальності вибірки за цим стовпцем,  $k$ , до кардинальності цілої таблиці,  $C$ , яке виражається у процентах:

- де  $k$  – кількість різних значень в стовпці, а  $C$  – кількість рядків в таблиці.

- Параметри  $k$  та  $C$  завжди включаються в статистику таблиці, тому оптимізатор може оперувати ними при побудові плану запиту.

$$S = \frac{k}{C} 100\%$$

# Випадки застосування некластерних індексів

---

- Запит повертає невелику кількість рядків (через фільтрування) і індексований стовпець має хорошу селективність (більше 95%).
  - Якщо селективність менша 95%, то скоріш за все оптимізатор не буде використовувати для виконання операції пошуку некластерний індекс, який базується на цьому стовпці.
  - Змінити поріг селективності не можна, але можна явно через підказку наказати оптимізатору використати індекс.
-

# Випадки застосування некластерних індексів

---

- Фрази WHERE та ORDER BY визначені для одного і того ж стовпця.
  - У цьому випадку некластерний індекс має подвійну перевагу: прискорює пошук потрібних записів і їх сортування, оскільки вибрані дані вже посортовані.
  - Зауважимо, що для **великої кількості результатних рядків більш корисними є кластерні індекси.**
-

# Випадки застосування некластерних індексів

---

- ❑ При використанні операції з'єднання таблиць (JOIN) та групування (GROUP BY).
    - Необхідно створити некластерні індекси для стовпців, які беруть участь в операціях з'єднання і групування
  - ❑ Стовпець або стовпці, які будуть проіндексовані, є широкими.
    - Хоча створення індексів на широкі стовпці є в принципі неефективним способом, то у випадках, коли немає інших варіантів оптимізації, використовують некластерні індекси.
-



# Робота з некластерними індексами

---

- Навіть коли стовпець має некластерний індекс, оптимізатор не завжди використовує його в операції пошуку.
    - Замість цього виконує сканування таблиці або кластерного індексу, що знижує ефективність виконання запиту.
    - Така ситуація виникає при кореляції (прямому зв'язку) даних
  - Наприклад, кластерний індекс створено для стовпця, що містить дату операції, а некластерний – для стовпця з номером операції. У цьому випадку є кореляція між датами, які зростають, і відповідними номерами замовлень, які зростають. Оптимізатор працює у припущенні, що немає кореляції. І це веде до помилкових рішень.
-

# Способи керування оптимізацією

---

- ❑ необхідно перевпорядкувати складений некластерний індекс так, щоб на першому місці був найбільш селективний стовпець
  - ❑ створити покриваючий індекс
  - ❑ дати підказку оптимізатору
-

# Покриваючі індекси (Covering Indexes)

---

- ❑ **Некластерні складені індекси** корисні тоді, коли вони є покриваючими індексами.
  - ❑ Продуктивність підвищується у тих випадках, коли індекс містить усі стовпці запиту.
    - У цьому випадку оптимізатор може знайти усі значення стовпців всередині цього індексу.
    - Тоді відпадає необхідність звертатись до таблиць або даних кластерних індексів, в результаті чого знижується інтенсивність дискових операцій введення / виведення.
    - Для розширення покриття стовпців не потрібно створювати широкий ключ індекса, а треба використовувати індекс з включеними стовпцями.
-

# Побудова покриваючого індексу

---

- Треба включити в некластерний індекс усі стовпці, які зустрічаються у фразах SELECT, JOIN і WHERE.
  - Корисними можуть бути такі індекси і в операціях групування GROUP BY.
  - **Якщо таблиця має кластерний індекс, то стовпець (або стовпці) кластерного індексу автоматично додається вкінець кожного некластерного індексу таблиці.** Це може призвести до виникнення покриваючого індексу без вказання стовпця кластерного індексу у визначенні некластерного індексу.
    - Наприклад, таблиця має кластерний індекс стовпця **A**, тоді некластерний індекс, визначений на стовпцях **B** і **C**, буде мати ключові значення стовпців **B**, **C** і **A**.
-

# Відфільтровані індекси

---

- ❑ **Відфільтрований індекс** – це оптимізований некластерний індекс, який підходить для запитів, які виконують вибірку з наперед визначеного набору даних.
  - ❑ Він використовує предикат фільтрації для індексування частини рядків в таблиці.
  - ❑ У створенні такого індексу використовується фраза WHERE.
  - ❑ Добре спроектований відфільтрований індекс дозволяє оптимізувати запити, знизити затрати на обслуговування та зберігання індексів у порівнянні з повнотабличними некластерними індексами.
-

# Випадки застосування відфільтрованих індексів

---

- Якщо у стовпця є мало унікальних значень, наприклад лише 0 і 1, то при виконанні більшості запитів некластерний індекс, побудований на цьому стовпці, не буде використовуватись.
  - Для такого типу даних можна створити відфільтрований індекс для окремого значення, яке з'являється в невеликій кількості рядків.
    - Наприклад, якщо більшість значень рівне 0, то оптимізатор може використати відфільтрований індекс для рядків даних, в яких міститься значення 1.
-

# Випадки застосування відфільтрованих індексів

---

- Стовпці містять добре визначені підмножини даних, вказані в інструкціях SELECT. Наприклад:
    - розріджені стовпці, які містять невелику кількість неNULL значень;
    - різнорідні стовпці, які містять категорії даних;
    - стовпці, які містять діапазони значень, такі як сума грошей, відрізки часу і дати;
    - секції таблиці, які визначені логікою простого порівняння для значень стовпців.
-

# Правило 1 використання відфільтрованих індексів

---

- Зниження затрат на обслуговування відфільтрованого індексу є більш наглядним, коли кількість рядків в індексі є невеликою у порівнянні з повнотабличним індексом.
  - Якщо відфільтрований індекс включає велику кількість рядків, його обслуговування буде більш затратним у порівнянні з повнотабличним, і його використання є недоцільним.
-



# Правило 2 використання відфільтрованих індексів

---

- Відфільтровані індекси визначаються на одній таблиці і підтримують лише прості оператори порівняння.
  - Якщо виникає необхідність у критерії фільтру, який посилається на кілька таблиць або має складну логіку, то треба створювати віртуальну таблицю.
-

# Рекомендації щодо написання запитів на вибірку даних

---

- Іншим способом оптимізації виконання запитів, крім використання індексів, є написання коректних запитів на вибірку з
    - оптимізацією умов
    - оптимізацією сортування
    - оптимізацією групування і агрегування
    - оптимізацією з'єднання таблиць
-

# Оптимізація умов

---

- ❑ Пошук даних за умовою (фраза WHERE) використовується найчастіше
  - ❑ Оптимізуючи умови, можна досягнути значної продуктивності запитів
  - ❑ Чим простіші (коротші) операнди використовуються в операціях порівняння, тим вища швидкість виконання запиту
-

# Оптимізація умов. Оператори порівняння

---

впорядковані за швидкістю  
виконання, починаючи з  
найшвидшого:

☐ =

☐ >, >=, <, <=

☐ LIKE

☐ <>

---

# Оптимізація умов. Логічні оператори.

---

- ❑ Якщо в умові пошуку використовуються декілька **кон'юнкцій AND**, то при відсутності відповідних індексів і статистики такі вирази будуть виконуватись зліва направо, причому ніякі круглі дужки не зможуть змінити такий порядок.
  - ❑ Принцип перевірки послідовності кон'юнкцій такий, що якщо перший вираз є хибним, то решта перевірятись не будуть.
  - ❑ Тому доцільно першими розміщати вирази, імовірність істинності яких є малоімовірною.
  - ❑ Якщо для пари виразів імовірність приблизно однакова, то першим треба розмістити найпростіший.
  - ❑ Зауважимо, що такі рекомендації не відносяться до СУБД Oracle, де умови починають перевірятись з кінця. Відповідно, порядок виразів повинен бути протилежним.
-

# Оптимізація умов. Логічні оператори.

---

- ❑ Ситуація з **оператором OR** є протилежною до кон'юнкції.
  - ❑ Вирази в умові повинні бути розміщеними у порядку спадання імовірностей – від найбільшої.
  - ❑ Фірма Microsoft рекомендує використовувати цей метод при побудові запитів.
  - ❑ Це не відноситься до СУБД Oracle, де умови з диз'юнкцією повинні розміщуватись за зростанням імовірності істиності.
-

# Оптимізація умов. Логічні оператори.

---

- При використанні в умовах пошуку **поєднання AND і OR** можна використати розподілений закон:

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$$

- Дослідним шляхом встановлено, що вираз зліва виконується швидше.
  - Деякі СУБД самі вміють оптимізувати запити такого типу. Але краще відразу записати оптимальний вираз.
-

# Оптимізація умов. Логічні оператори.

---

- **Операцію NOT** завжди потрібно приводити до читабельного вигляду.
    - Наприклад, умову  
WHERE NOT (column1>5)  
відразу можна записати:  
WHERE column1<=5.
  - Складніші умови можна записати, використовуючи правило де Моргана:  
NOT(A AND B)=(NOT A) OR (NOT B);  
NOT(A OR B)=(NOT A) AND (NOT B).
    - Наприклад, умову  
WHERE NOT (column1>5 OR column2=7)  
можна перетворити на простішу:  
WHERE column1<=5 AND column2<>7.
-



# Оптимізація умов

---

- **Предикат *IN*** працює **набагато швидше** від серій диз'юнкцій OR.
    - Тому необхідно завжди замінити OR на IN, де це можливо, не зважаючи на те, що деякі СУБД самі роблять таку оптимізацію.
  - Якщо ж є вибір використання ***IN*** чи ***BETWEEN***, то ефективніше використовувати BETWEEN.
    - Наприклад, умова  
WHERE column1 IN (1,3,4,5)  
оптимізується до вигляду:  
WHERE column1 BETWEEN 1 AND 5  
AND column1<>2
-

# Оптимізація умов

---

- Якщо в умові пошуку часто використовується пошук за шаблоном з використанням **оператора LIKE** і через це виникають проблеми з швидкістю виконання запитів, то необхідно розглянути можливість використання служби повнотектового пошуку MicrosoftSearch, яка базується на full-text індексах і працює набагато швидше.
-

# Оптимізація умов і використання індексів

---

- ❑ Основний спосіб підвищення швидкодії фрази WHERE – застосувати індекси для організації пошуку.
- ❑ Однак, це ще не гарантує, що вони будуть використані, оскільки не всі логічні предикати операцій порівняння допускають використання індексів.
- ❑ Індекси, як правило **не застосовуються** для виконання операцій пошуку з такими предикатами: IS NULL, <>, NOT, NOT EXIST, NOT IN, NOT LIKE.
- ❑ Не може використовувати індекс оператор LIKE, який починається з символу підстановки (маски).
  - Наприклад, умова пошуку  
column LIKE 'П%'  
має потенційну можливість використати індекс, а умова  
column LIKE '%П'  
такої можливості не має.
- ❑ Хоча можна примусити сервер застосувати індекс і в цьому випадку через підказку оптимізатору.

# Оптимізація сортування

---

- ❑ Чим більший об'єм даних, тим більше часу займе сортування.
  - ❑ На швидкість сортування впливає три фактора:
    - кількість вибраних рядів;
    - кількість стовпців, вказаних у фразі ORDER BY;
    - довжина і тип стовпців, вказаних у фразі ORDER BY.
  - ❑ Альтернативою такій оптимізації є використання індексів.
-

# Оптимізація групування

---

- ❑ Необхідно використовувати якомога менше стовпців групування.
  - ❑ І якщо у фразі HAVING не використовується агрегатна функція, то вважається, що вона є еквівалентною до фрази WHERE, яку і потрібно використовувати для оптимізації запиту.
  - ❑ Однак, необхідно враховувати специфіку СУБД, оскільки в багатьох СУБД фрази WHERE і HAVING не є рівноцінними і виконуються **не однаково**.
  - ❑ Якщо відбувається групування без агрегатних функцій, доцільно використовувати оператор DISTINCT.
-

# Оптимізація групування і агрегування

---

- При використанні агрегатних функцій MIN і MAX необхідно враховувати, що виконуються вони швидше, якщо записані окремо.
    - Це означає, що їх краще використовувати в різних запитах або в запитах з використання операції об'єднання UNION.
  - При використанні функції SUM:
    - оптимальнішим буде вираз  $SUM(x+y)$ , а не  $SUM(x)+SUM(y)$ .
    - для віднімання – навпаки:  $SUM(x)-SUM(y)$  обробляється швидше, ніж  $SUM(x-y)$ .
-

# Оптимізація з'єднання таблиць

---

- ❑ Операція з'єднання таблиць (JOIN) є найдорожчою операцією реляційної алгебри, не враховуючи операції декартового добутку.
  - ❑ Швидкість виконання таких операцій залежить від організації самої таблиці, використання первинного та зовнішніх ключів.
  - ❑ Однозначних рекомендацій як використовувати різні способи з'єднання таблиць не існує. Все залежить від конкретного випадку і архітектури бази даних.
-

# Оптимізація з'єднання таблиць

---

- Якщо необхідно регулярно виконувати операції з'єднання таблиць, то ефективність в загальному випадку буде оптимальною при з'єднанні стовпців однакового типу, кожен з яких має власний індекс. При цьому вважається, що
    - кластерні індекси кращі некластерних,
    - чим вища селективність стовпців, за якими виконується з'єднання, тим краще.
  - На швидкість з'єднання впливає тип даних полів зв'язку. Вважається, що найшвидше виконується з'єднання для полів з цілочисельними типами даних.
-



# Оптимізація з'єднання таблиць

---

- Якщо запит вимагає з'єднання більше чотирьох таблиць, необхідно подумати про денормалізацію бази даних або про створення індексованої віртуальної таблиці, яка реалізує попередні з'єднання.
- Необхідно уникати використання \* для вибору усіх стовпців в обох таблицях.
- Ефективність з'єднання буде кращою, якщо фізичні сторінки даних заповнені максимально щільно. Це можна досягнути за допомогою:
  - задання високого значення параметру FILLFACTOR для індексних сторінок,
  - шляхом оптимізації типів даних і розмірів стовпців у таблицях.
- Необхідно вчасно перебудовувати індекси.

# Типи операторів з'єднання в SQL Server при виконанні запитів

---

- ❑ NESTED LOOP JOIN – з'єднання блоками з використанням вкладених циклів (використовуючи або не використовуючи індекси);
- ❑ MERGE JOIN – з'єднання сортуванням-злиттям;
- ❑ HASH JOIN – з'єднання через використання функції перемішування (хеш-функції).

Однозначних рекомендацій як використовувати різні способи з'єднання таблиць не існує. Все залежить від конкретного випадку і архітектури бази даних. Ознайомитись з алгоритмами цих способів з'єднання можна у відповідній літературі.

---

# Оптимізація підзапитів

---

- ❑ Використання підзапитів є близькою альтернативою до операції з'єднання таблиць.
  - ❑ Основна проблема для їх оптимізації – не оптимізація самого коду, а вибір правильного способу для реалізації запиту.
  - ❑ Існують рекомендації, коли доцільніше використовувати підзапити, а коли з'єднання таблиць.
  - ❑ Основною перевагою використання оператора JOIN полягає в тому, що не потрібно вказувати СУБД яким способом виконувати з'єднання таблиць.
  - ❑ Основна перевага підзапитів така, що цикл підзапиту може мати декілька ітерацій, що й може покращити продуктивність виконання запиту.
-

# Переваги з'єднання таблиць по відношенню до підзапитів

---

- ❑ якщо запит містить фразу WHERE, то оптимізатор при використанні з'єднання оцінює запит в цілому, а у випадку підзапитів, вони будуть оптимізовані окремо (по частинах);
  - ❑ деякі СУБД (наприклад, Oracle) ефективніше працюють з операціями з'єднання таблиць;
  - ❑ після з'єднання в результатній таблиці виводиться вся вказана інформація, а в підзапитах частина даних використовується як параметр в умові пошуку.
-

# Переваги підзапитів по відношенню до з'єднання таблиць

---

- ❑ підзапити допускають використання більш вільних, різноманітних умов
  - ❑ підзапити можуть містити фрази GROUP BY, HAVING, які набагато складніше оптимізувати в операціях з'єднання таблиць
  - ❑ засоби оптимізації сучасних СУБД значно покращились
-

Дякую за увагу

---