РОЗДІЛ 3. ОСНОВИ ПРОЕКТУВАННЯ РЕЛЯЦІЙНИХ БАЗ ДАНИХ

3.1. Реляційна модель(relation model)БД є найбільш популярною в наш час, і більшість СКБД орієнтовані саме на неї. Вперше концепція реляційної моделі бази даних запропонована Е.Ф.Коддом(на той час він працював в корпорації IBM) в 1970 році для забезпечення незалежності представлення та опису даних від прикладних программ. В 1979 році Кодд зробив спробу усунути недоліки власної основоположної роботи і опублікував розширену версію реляційної моделі — RM/T (1979), потім ще одну версію — RM/V2 (1990).

Основою РМБД є *відношення(relation)* - це двовимірна таблиця. Рядки таблиці називаються *кортежами*, а стовпці *атрибутами*. Терміни відношення, кортеж, атрибут прийшли з реляційної алгебри, на основі якої була побудована дана модель. Програмісти полюбляють вживати терміни *файл, запис* і *поле* замість відношення, кортеж та атрибут, решта користувачів – таблиця, рядок, стовпчик. Щоб таблиця була відношенням, вона має відповідати наступним вимогам – значення в клітинах мають бути однозначними та мати однаковий тип для визначеного стовпчика. Атрибути повинні мати унікальні імена в межах таблиці. Рядочки та стовпчики не впорядковані, не має двох однакових рядків. Кількість кортежів в таблиці називається *кардинальним числом,* кількість атрибутів - *стелінню*.

Для відношення визначають ідентифікатор (один, або кілька стовпчиків). Ідентифікатор, значення якого є унікальними називають *первинним ключем*. За значення первинного ключа можна однозначно визначити будь-який рядок таблиці. *Домен* - множина допустимих однорідних значень деякого атрибута.

В РБД користувач вказує які саме дані йому потрібні, а не що з ними роботи(на відміну від інших моделей). Функції керування даними виконує СКБД. Для цього існує так званий *оптимізатор*, саме цей механізм визначає, як найефективніше відібрати дані для запиту користувача. Крім того, СКБД виконує ще й функції каталогу. В ньому зберігається опис всіх об’єктів, з яких складається СКБД. Дані в каталозі також представлені у вигляді

таблиць, тож для нього не потрібно якихось особливих засобів керування.

Отже, *реляційна база даних*(РБД) – це набір відношень, що зв’язані між собою одним із можливих типів зв’язку: один-до-одного (працівник-зарплата), один-до-багатьох (будинок-жильці) , багато-до-одного(студент-група) та багато-до-багатьох (студент-викладач). Зв'язок багато-до-багатьох напряму в РБД не підтримується. Крім перелічених між відношеннями можуть існувати ще й множинні зв’язки (студент-викладач:1-й зв'язок - заняття, 2-й-наукове керівництво).

В реляційних моделях розрізняють:

- *іменовані відношення* – це постійне відношення, що створюється в СКБД операторами створення, використовується для більш зручного представлення інформації користувача;
- *базове відношення* – є частиною БД, при проектуванні їм дають власні імена;
- *утворене відношення* – те, яке було визначене через інші, шляхом використання засобів СКБД;
- *представлення* – є фактично утвореним іменованим відношенням, виражається виключно через оператори СКБД, фізично в базі даних не існує;
- *результат запиту* – це іменоване утворене відношення, що містить дані, яке є результатом запиту користувача, в базі не зберігається;
- *збережені відношення* – те, що зберігається в пам’яті ПК, до них, як правило, відносяться базові.

Особливе місце в реляційних базах даних відводиться *цілісності даних*. Щоб розібратись з цим поняттям, визначимо з поняттям *ключ* . Кожне відношення має хоча б один *можливий ключ*. Якщо він один, він стає первинним, якщо кілька, первинним вибирають будь-який. При виборі первинного ключа перевагу віддають такому, що складається з мінімального числа атрибутів.

Кожне відношення має хоча б один можливий ключ(або ключ-кандидат). Один з ключів приймається за первинний, тобто такий за допомогою якого можна однозначно визначити будь-який кортеж таблиці. При виборі первинного ключа перевагу віддають таким ключам, що складаються з одного атрибута, небажано

використовувати ключі з довгими текстовими значеннями. Найчастіше вибирають ключі з цілочисельними атрибутами. Наприклад, табельний номер або номер паспорта. Неприпустимо, щоб первинний ключ приймав невизначені значення.

Розглянемо тепер поняття *зовнішнього ключа*. Якщо відношення С пов'язане з відношеннями А і В, то воно повинне включати зовнішні ключі, які відповідають первинним ключам відношень А і В, як відображено на Рис. 3.1.

При цьому потрібно вирішити питання, якими мають бути зовнішні ключі, та чи можлива поява невизначених значень в зовнішніх ключах. Іншими словами : чи можливе існування кортежу у відношенні для якого немає відповідного кортежу у зв'язаному відношенні. Ще потрібно вирішити, що буде відбуватись у разі знищення кортежів в одному з пов'язаних відношень.

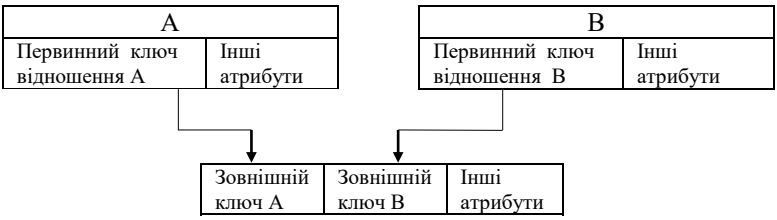


Рис.3.1. Зв'язки між відношеннями

. Для зовнішнього ключа потрібно визначити проблему можливості(або неможливості) прийняття зовнішнім ключем. невизначених чи нульових значень. Іншими словами, чи може існувати кортеж у відношенні, для якого невідомий кортеж у пов'язаному із ним відношенні. Необхідно також передбачити, що відбуватиметься, при знищенні кортежів, на які посилається зовнішній ключ. При цьому розглядаються наступні можливості:

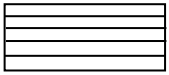
- Операція *каскадування* - знищення кортежів призводить до знищення пов'язаних даних. Наприклад, знищення інформації про працівника призводить до знищення інформації про його зарплату;
- Операція обмежується, тобто можна знищити лише ті кортежі, для яких немає пов'язаних з ними даних.

Те саме обумовлюють для операції поновлення первинного ключа

- каскадування – при зміні значення первинного ключа, міняється значення в пов'язаних кортежах інших таблиць;
- обмеження – можна змінити значення первинного ключа лише для непов'язаних з ним кортежів в інших таблицях.

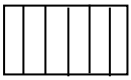
Формальною основою РБД є реляційна алгебра(РА), яка базується на теорій множин де розглядаються спеціальні оператори над відношеннями. Основних операторів в реляційній алгебрі 8. Вона замкнута відносно поняття відношення, тобто після виконання будь-якої операції РА над відношенням ми отримуємо відношення.

Опишемо варіант РА, що був запропонований Е.Ф.Коддом:

1. Операція *вибірки.* Результатом виконання операції вибірки є відношення, що включає лише ті кортежі, що відповідають заданій умові. Схематично це можна відобразити наступним чином.
2. 

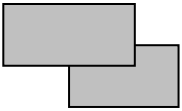
Тобто з відношення будуть відібрані лише ті кортежі, що відповідають заданій умові. Іншими словами: оператор вибірки виділяє горизонтальну підмножину рядків. Наприклад, вивести список студентів, що навчаються на 1 курсі

3. Операція *проекція.* Результатом цієї операції є відношення, яке містить кортежі вихідного відношення, для заданого набору атрибутів. Тобто, З вихідного відношення вибирається деяка множина стовпців. Схематично це можна відобразити наступним чином.

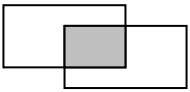


4. Операція о*б'єднання*. Результатом операції є відношення, яке містить всі кортежі, як першого так і другого відношення. Щоб ця операція мала зміст, відношення мають бути сумісними по об'єднанню, тобто мати однакову кількість атрибутів, що належать тому самому домену, тобто якщо 3-й атрибут першого відношення –

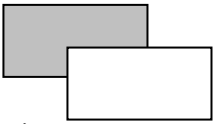
належить домену Прізвище, то 3-й атрибут другого відношення також має належати до того ж домену



5. Операція *перетин*. Результатом буде відношення, що містить кортежі, які належать як 1-му так 2-му відношенням, при цьому відношення мають бути сумісні по об'єднанню.



6. Операція *різниця*. Результатом є відношення, що містить лише кортежі, що входять в 1-ше відношення, і жоден не входить в 2-ге.



Добуток. В результаті виконання цієї операції отримаємо відношення, кортежі якого є комбінацією кортежів обох відношень.

А
Б
В

Г
Д

⇨

А	Г
А	Д
Б	Г
Б	Д
В	Г
В	Д

З'єднання двох відношень за деякою умовою є відношення, кортежі якого є комбінацією відношень , що задовольняють заданій умові.

А	Г
Б	Г
В	Д

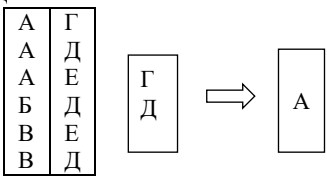
Г	Ж
Д	З
Е	И

⇨

А	Г	Ж
Б	Г	Ж
В	Д	З

Операція реляційного *ділення* має 2 операнди – бінарний та унарний. Результатом цієї операції є відношення що складається з кортежів, що включають значення першого атрибуту 1-го

відношення, такі що значення 2-го атрибуту співпадають з набором значень 2-гог відношення.



Крім перелічених операцій є ще операція *перейменування*, в результаті якої отримаємо відношення значення кортежів якого співпадають з вихідними, але імена атрибутів інші.

Операція *присвоювання* - зберігається результат обчислень виразу у існуючому відношенні БД.

Реляційне числення є основою для формування запитів до БД. Припустимо, в нас є відношення СТУДЕНТИ(Номер, ФІО, стипендія, номер_гр) та ГРУПИ(номер_гр, кілк_ст, стар_гр). Потрібно дізнатись прізвища студентів, що є старостами групи, в яких групах де кількість студентів більше 25. Якби для формування такого запиту використовувалась лише РА ми отримали б приблизно наступне:

- виконати з'єднання відношень СТУДЕНТИ і ГРУПИ за умовою СТУДЕНТИ.номер_гр= ГРУПИ.номер_гр:
- організувати вибірку з отриманого відношення за умовою кілк_ст>25:
- здійснити проекція для відношення, що було отримане в попередній операції за атрибутами ФІО, номер.

В реляційному численні це виглядає наступним чином :

вивести ФІО, номер з таблиць СТУДЕНТИ,ГРУПИ для яких ГРУПИ.кілк_ст >25 та СТУДЕНТИ.номер=ГРУПИ.стар_гр .

Оптимізатор СКБД сам вирішує в якому порядку та як виконувати ці операції.

Реляційне числення є тим математичним апаратом на якому побудована структурована мова запитів SQL, що стала основою всіх сучасних СКБД..

Проектування логічної структури бази даних

Бази даних є моделями, що відображують реальний світ з точки зору тієї моделі, яка є в уяві користувача. Яку модель краще обрати

для проектування логічної структури бази даних – справа розробника.

Реляційна модель важлива з двох причин. По –перше, конструкції РМБД мають широкий і загальний характер і є незалежними від всіх СКБД. По-друге, реляційна модель покладена в основу майже всіх СКБД. Отже розуміння принципів цієї моделі є актуальним для всіх категорій користувачів баз даних..

Ми з'ясували, що основним елементом РМБД є таблиця. Тому проектування БД починають з формування структури таблиць. Визначають всі атрибути, що мають зберігатись в БД. Потім формують таблиці. Можна звичайно помістити все в одну таблицю, але такі дані не завжди зручно обробляти. Структура такої таблиці може бути незручною для модифікації даних, є небезпека надлишкового повторення даних. Це приводить до неекономного використання пам'яті комп'ютера при збережені даних. Виникає питання: як правильно сформувати структуру таблиць, щоб дані можна було швидко і зручно обробляти.

Розглянемо кілька прикладів неефективної структури таблиць.

прізвище	секція	оплата
Іванов	лижі	100
Василів	футбол	150
Коник	баскетбол	110
Козак	футбол	150
Мисько	баскетбол	110

Знищення рядка 1, приведе до втрати інформації, про оплату за секцію ЛИЖІ. Ця проблема називається *аномалією модифікації*. В даному випадку – аномалія знищення, Якщо ж ми захочемо внести дані про секцію підводного плавання, то не зможемо це зробити, поки в цю секцію не запишеться хоча б одна людина. Такий недолік має назву – *аномалія вставки*. Для уникнення таких проблем, одну таблицю розбивають на кілька В нашому прикладі потрібно створити 2 таблиці СЕКЦІЯ(назва, сума_оплати) та ВІДВІДУВАЧІ(назва_секції, Прізвище). Таким чином ми позбулися вищезгаданих аномалій. .При розбитті потрібно слідкувати щоб не порушувалися умови цілісності БД. Тобто студент не може записатись в неіснуючу секцію.

Процес перетворення відношень що мають деякі недоліки і відношення що їх не мають називається ***нормалізацією***.

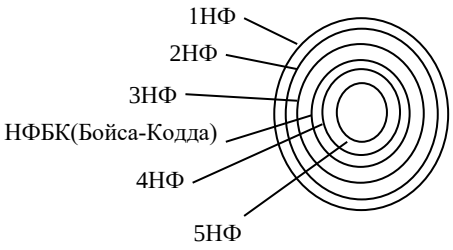
Класи відношень та способи уникнення аномалій називаються ***нормальними формами.***

Термін цей був введений Коддом (1970 р.), саме він дав визначення різним нормальним формам відношень. За теорією Кодда всього є шість нормальних форм. В залежності від своєї структури відношення може бути в 1-й, 2-й чи іншій нормальним формам.

Потрібно відмітити, що термін *нормальна таблиця* можна сприймати не лише в рамках математичного апарату, але й просто так. Тому що нормальні таблиці справді є нормальними (і навіть хорошими), в тому змісті, що при роботі з ними ві труднощі практично зникають а час обробки даних значно скорочується.

Офіційне визначення є наступним: *нормальна форма це вимоги, що накладаються на структуру таблиці в теорії реляційних баз даних для уникнення надлишковості функціональних залежностей та аномалій модифікації.*

Як показано на Рис. 3.1. нормальні форми є вкладеними, тобто відношення в 2-й нормальній формі є також відношенням в 1-й нормальній формі, відношення в 5-й нормальній формі є одночасно відношенням в 4-й, НФБК, 3=й, 2-й, та 1-й нормальних формах. Окремо стоїть лише доменно-ключова нормальні форма.



*Доменно-ключова нормальна форма.

Рис.3.1. Взаємозв’язок нормальних форм

Нормальні форми з 1-ї по 5-ту допомагали покращити структуру таблиць, але не було теорії яка гарантувала, що якась з цих форм знищить всі аномалії. Лише в 1981 році коли Р.Фагін ввів ДНКФ було виявлено, що відношення в ДНКФ вільне від усіх аномалій модифікації. Більше того, будь-яке відношення, що не має аномалій модифікації знаходиться в ДНКФ.

До введення ДНКФ теоретики реляційних баз даних віднаходили все нові і нові аномалії модифікації та способи їх уникнення. Лише доведення Р.Фагіна спростило ситуацію. Якщо ми можемо привести відношення в ДНКФ, то можна бути впевненим, що воно не має аномалій модифікації, питання лише в тому, як привести відношення до ДНКФ.

Але повернемось до нормальних форм з 1-ї по 5-ту, та дамо визначення кожній з них.

Для початку розглянемо поняття функціональної залежності.

Функціональні залежності – це зв'язок між атрибутами. Наприклад за номером клієнта можна визначити стан його рахунку, тоді кажуть що атрибут стан рахунку функціонально залежить від номера клієнта. Отже, *атрибут У функціонально залежить від атрибута Х, якщо значення атрибута Х визначає значення У*. Х - є детермінантом, але не обов'язково є ключем. Якщо атрибут *Вартість =Ціна*Кількість*, то кажуть що *Вартість* функціонально залежить від атрибутів *Ціна* та *Кількість*.

Отже, <u>*перша нормальна форма(1НФ)*</u>

Будь-яка таблиця(відношення) знаходиться в 1НФ, тобто атрибути – унікальні, рядки та стовпчики не впорядковані, немає однакових рядків.

Розглянемо перший приклад – відношення СЕКЦІЯ. Таблиця знаходиться в 1НФ. Ключем є комбінація атрибутів : (прізвище,секція) Але атрибут *Оплата* залежить від атрибуту *секція*, тобто не від всього ключа а лише від частини. Наша таблиця має недоліки: аномалії знищення та вставки, функціональні залежності від неповного ключа, тому потребує покращення. Ми повинні розділити її на 2 таблиці: СЕКЦІЯ(назва, сума_оплати) та ВІДВІДУВАЧІ(назва_секції, Прізвище), щоб тим самим позбутися вищезгаданих проблем. Можна сказати, що ми звели наші відношення до 2-ї нормальної форми.

Відношення знаходиться в 2НФ, якщо всі його не ключові атрибути залежать від цілого ключа.
Розглянемо такий приклад:

Номер студента.	гуртожиток	плата
100	3	550
110	5	580
120	3	550
150	7	600

Поле ***Номер студента*** – ключ. За номером студента можна визначити гуртожиток в якому він проживає, а за номером гуртожитку – плату. Отже, непрямим чином за полем ***Номер студента*** можна визначити плату за гуртожиток.

Номер студента -.> гуртожиток, гуртожиток -.> плата

Таку залежність називають транзитивною. В нашому прикладі:відношення знаходиться в 2НФ, але має транзитивні залежності. При роботі з таким відношенням також виникають пеані проблеми: надлишкове повторення інформації, аномалії знищення. Наше відношення потребує покращення і приведення до 3-ї нормальної форми.

Отже, відношення знаходиться в 3НФ, як що воно знаходиться в 2НФ та не має транзитивних залежностей.

Для приведення в 3НФ, слід розбити нашу таблицю на 2:
СТУДЕНТИ(ном_студ, ном_гурт), ГУРТОЖИТКИ(ном_гурт, оплата).

Наступне відношення містить інформацію про

номер студента	предмет	викладач
100	алгебра	Кот А.П.
110	геометрія	Іванченко В.І.
120	програмування	Козаченко А.А.
150	алгебра	Кот А.П.
200	програмування	Самсонюк В.В.

Відношення відповідає умовам 1НФ, 2НФ (кожен атрибут визначається повним ключем), 3НФ(не має транзитивних залежностей). Але не занходиться в НФБК.
Відношення знаходиться к НФБК, якщо кожний детермінант є ключем кандидатом
Для нашого прикладу:
Ключ (номер студента + предмет) – первинний
Ключ (номер студента + викладач) – вторинний(ключ кандидат)
Викладач -> предмет(викладач - детермінант)
Один предмет можуть вести різні викладачі.
Викладач може бути вести лише по одному предмету.
Існує аномалія - якщо знищити запис про студента з номером 110, ми втратимо інформацію, що в Іванченко викладає геометрії.
Зрозуміло що відношення потребує покращення, розділивши його на 2 ми отримаємо НФБК.

Відношення знаходиться в 4НФ, якщо воно є в НФБК, та не має багатозначних залежностей.

Наступне відношення відповідає умовам 1НФ,2НФ,3НФ,НФБК, але не відповідає визначення 4НФ

номер студента	спеціальність	секція
100	менеджмент	лижі
100	прикладна математика	футбол
120	бухгалтерський облік	баскетбол
150	менеджмент.	футбол
120	прикладна математика	баскетбол

Припустимо, студеним м. вчитись на кількох спеціальностях та займатись в кількох секціях.
Ключ – (номер студента ,спеціальність,секція) – первинний ключ.
Між атрибутами номер студента –>> секція, Номер_ст.–>> спеціальність(–>>багатозначні залежності). Незручність -

багаторазове повторення інформації(надлишковість). Розділивши відношення на 2 ми зведемо його до 4НФ.

У всіх попередніх прикладах ми , для покращення обробки відношень та забезпечення відповідності вимогам певної нормальної форми розділяли одне відношення на 2. Ви являється, що іноді такі декомпозиції не можна виконати без певних втрат,але можна виконати розподіл на 3 та більше таблиць. П'ята форма закликає, щоб всі можливі декомпозиції були виконані

П'ята НФ на практиці практично не використовується. Вона пов'язана із залежностями, що мають дещо невизначений характер. Мова іде про відношення,які можна розділити на кілька менших, але потім важко відновити попередню інформацію.

Доменно-ключова нормальна форма (ДКНФ)

Всі вищезгадані нормальні форми були виділені дослідниками, що виявили аномалії в деяких відношеннях, які знаходились в нормальній формі нищого порядку. Хоч кожна наступна НФ вирішувала частину проблем попередньої, ніхто не міг сказати, які ще проблеми будуть виявлені. Тепер ми розглянемо форму, яка вільна від аномалій будь-якого типу.

В 19*1 р. Фагин опублікував статтю, в який він дав визначення доменно-ключовій нормальній формі. Він показав, що відношення в ДКНФ не лише немає аномалій модифікації, навпаки, якщо відношення не має аномалій модифікації воно є в ДКНФ.

Відношення є в ДКНФ якщо, кожне обмеження що накладається на відношення є логічним наслідком визначення доменів та ключів.

Фагін визначає *обмеження, як будь-яке правило, що регулює можливі статичні значення атрибутів*(правила редагування, обмеження взаємовідносин и структури відношень, функціональні залежності, багатозначні залежності). Але, Фагин виключає з обмежень правила, що накладають зміни на значення. Наприклад, такі як «зарплата продавця за поточний період не може бути меншою ніж за попередній період».

Ключ – унікальний ідентифікатор кортежу. Домен – опис всіх припустимих значень атрибуту.

Інакше кажучи, відношення знаходиться в ДКНФ, якщо визначення всіх обмежень на домени і ключі приводить до виконання всіх обмежень. Більш того , оскільки відношення в ДКНФ не можуть мати аномалій модифікації, то СКБД може попередить їх виникнення виконуючи обмеженняо, що накладені на домени та ключі.

Нажаль жодного алгоритму приведення відношення до ДКНФ не існує. Знаходження ДКНФ є швидше мистецтвом ніж наукою.

Приклад:

наведений приклад демонструє доменно-ключову нормальну форму. Розглянемо обмеження, що накладені на відношення

СТУДЕНТ-КЕРІВНИК(ном_ст, ПІБ_ст, номер_викл, ПІБ_викл., співр_аспір).
Це відношення містить інформацію про студента та керівника його наукової роботи
Ключ: ном_ст
Обмеження ном_ст визначає ПІБ_ст,
ПІБ_ст визначає ном_ст
номер_викл та ПІБ_викл визначає співр_аспір
Тільки співробітники аспірантури можуть бути керівниками аспірантів.
Номер співробітника починається з 1, номер студента не може починатись з 1,
Ном_ст, для аспірантів починається з 9.
Співр_асп = 1, для співробітників аспірантури та 0 для інших викладачів.

Визначення доменів:

Номер_викл	CDDD, де C=1, D- будь-яке десяткове число
ПІБ -	CHAR(30)
Співр_аспір	[0,1]
Ном_ст	CDDD, де C<>1 C<>9, D- будь-яке десяткове число

Ном_ст(для аспіранта) CDDD, де C=9, D- будь-яке десяткове число

Визначення відношень і ключів
ППС(номер_викл, ПІБ_викл)
Ключ – номер_викл.

Керівник –аспірант (номер_асп,ПІБ, прізвище)
Ключ – номер_асп.
Кервник –студент(ном-ст, ПІБ, прізвище)
Ключ- ном_ст.

Синтез відношень

В попередніх прикладах ми розглядали процес проектування бази даних і аналітичних позицій. Вирішувалось питання: чиє таблиця зручною для обробки дпних, чи має вона аномалії модифікації.

Тепер підійдемо до питання проектування реляційної структури з іншого боку. Нехай в нас є набір атрибутів з певними функціональними залежностями. Як слід формувати з них відношення .

Перш за все згадаємо, що атрибути можуть мати залежності трьох видів

- A ->B і B ->A, зв’язок «один до одного»
- A ->B або B ->A «один до багатьох»
- A та B функціонально не зв’язані між собою, але можуть бути присутній зв'язок «багато до багатьох»

Правила, на основі яких відбувається синтез відношень:

1. Якщо два атрибути мають зв’язок «один до одного», вони можуть з'явитись разом, як мінімум в одному відношенні, інші атрибути, які функціонально визначаються даними теж можуть бути в цьому відношенні.
2. Якщо два атрибути мають зв’язок один до багатьох», та А визначає В але В не визначає А, вони можуть входити в одне відношенні, А- ключ, будь-який атрибут, що функт. Залежить від А можна помістити в це відношення.

3. Якщо два атрибути A та B функціонально не зв’язані, присутній зв'язок «багато до багатьох». Ці атрибути можуть бути в одному відношенні, (A,B) – ключ, будь-який атрибут, що функт. залежить від , (A,B) можна помістити в це відношення.

Денормалізація.

Не завжди процес нормалізації буває доцільним. Трапляються випадки, коли таблиці, що отримані шляхом нормалізації не зручні для використання в базі даних.

.

Приклад:

ФАКУЛЬТЕТ(назва, декан, зам декана1, зам декана2, зам декана3).

Згідно теорії нормалізації цю таблицю потрібно розділити на дві: ФАКУЛЬТЕТ(назва_ф, ПІБ_декана, ПІБ_замдекана), ЗАМДЕКАНА(назва_ф,,ПІБ_замдекана)

Але в цьому випадну СКБД потрібно буде читати для вибору даних 3 рядки таблиці. Зрозуміло що це не зовсім зручно для організації запитів та обробки даних, отже варто залишити таблицю в початковому вигляді.

Контрольні запитання та завдання до розділу 3

1. Що таке відношення
2. Дати визначення ключа(первинний та зовнішній ключ)
3. Що таке нормалізація
4. Назвати основні операції реляційної алгебри
5. Дати визначення функціональній залежності
6. Доменно-ключова нормальна форма

РОЗДІЛ 4. СТРУКТУРНА МОВА ЗАПИТІВ SQL

4.1. *Історія розвитку SQL* При роботі з реляційними базами даних користувач повинен лише вказати список таблиць та умови, яким повинні задовольняти дані, що вибираються. Як виконати поставлену задачу самим ефективним сопором «вирішує» сама СКБД.

Важливою вимогою до СКБД є наявність потужної і простої мови для виконання всіх необхідних користувацьких операції. В останні роки такою загально прийнятою мовою стала мова реляційних баз даних SQL (Structured Query Language) – структурована мова запитів.

До появи SQL в СУБД (незалежно від того, на якій моделі вони ґрунтувалися) доводилося підтримувати принаймні три мови, в яких зазвичай було мало спільного:

- мова визначення даних (МВД), що використовувалась для визначення структур БД (зазвичай загальну структуру БД називають схемою БД);
- мова маніпулювання даними (ММД), що дозволяє створювати прикладні програми, які взаємодіють з БД;
- мова адміністрування БД (МАБД), за допомогою якої можна було виконувати службові дії (наприклад, змінювати структуру БД або виконувати її налаштування з метою підвищення ефективності).
- Крім того, якщо потрібно було надати користувачам СУБД інтерактивний доступ до БД, доводилося вводити ще одну мову, оператори якого виконуються в діалоговому режимі. Мова SQL дозволяє вирішувати всі ці завдання.

На початку 70 років XX століття Коддом було розпочато проект, що отримав назву System/R, в якому перевірялась та доводилась працездатність реляційної моделі. В результаті чого було зроблено висновок: реляційні моделі цілком дієздатні та можуть використовуватись для розробки програмних продуктів. В 1977 році була створена компанія Relation, Software Inc. Якою була розроблена СКБД основана на мові SQL – Oracle(1979 р). Дослідницька група компанії IBM створили прототип реляційної. СКБД – Ingres з мовою запитів QUEL. Комерційна версія Ingres з’явилась в 1981 році. В 1986 році QUEL був замінений на SQL. В

першій половині 80-х розробники РБД боролись за визнання свого продукту оскільки швидкодія традиційних архітектур БД була кращою. З 1986 року ця ситуація суттєво змінилась, з того часу SQL офіційно став стандартом розробки РБД. Тепер він є ключовою частиною проектування клієнт - серверних систем.

Перевагою мови SQL є наявність міжнародних стандартів. Перший міжнародний стандарт SQL був прийнятий в 1989 р. – SQL1(або SQL89), він використовувався в більшості тогочасних СКБД(Informix, Sybase, Ingres та інших). Але розвиток ІС вимагав розвитку можливостей SQL. Другий міжнародний стандарт з'явився в1992 р. – SQL2, а в 1993 р. – SQL3. Якщо різниця між SQL1 та SQL2 є кількісними, то SQL3 – відповідає якісними перетворенням: введені нові типи даних, механізм генераторів і тригерів, які відповідають об'єктній орієнтації. SQL-2003 - введено розширення для роботи з XML-даними, віконні функції (застосовувані для роботи з OLAP-базами даних), генератори послідовностей і засновані на них типи даних. SQL2006 - розширена функціональність роботи з XML-даними, з'явилася можливість спільно використовувати в запитах як SQL так і XQuery. SQL2008 - поліпшено можливості віконних функцій, усунуті деякі неоднозначності стандарту SQL2003.

Назва мови SQL (Structured Query Language - структурований мова запитів) тільки частково відображає його суть. Звичайно, мова завжди була головним чином орієнтований на зручну і зрозумілу користувачам формулювання запитів до реляційної БД, але насправді з самого початку задумувався як повний мова БД. Під цим ми розуміємо те, що (принаймні, теоретично) знання SQL повністю достатньо для виконання будь-яких осмислених дій з базою даних, керованої SQL-орієнтованої СУБД. Крім операторів формулювання запитів і маніпулювання БД мова містить:

- засоби визначення схеми БД і маніпулювання схемою;
- оператори для визначення обмежень цілісності і тригерів;
- засоби визначення уявлень БД;
- кошти авторизації доступу до відносин та їх полям;
- засоби управління транзакціями.

SQL орієнтований на операції с даними, що представлені у вигляді логічних, взаємозв'язаних між собою таблиць. Він

орієнтований на кінцевий результат обробки даних а не на процедуру їх обробки(як у випадку коли дані збережені у вигляді файлів). SQL сам визначає де знаходяться дані, індекси та які послідовності команд треба виконати, щоб найбільш ефективним чином отримати бажаний результат.

Мова SQL має певні стандарти, у випадку появи нової платформи, більш перспективної, об'єкт орієнтований на стандарти, може бути легше перенесений на неї, крім того програміст може бути впевнений, що стандарт буде реалізований у всіх СКБД.

SQL не можна в повній мірі віднести до традиційних мов програмування, вона не має традиційних операторів керування кодом програми, операторів опису типів та інше. Він має лише набір операторів доступу до даних, що зберігаються в БД.

З 1987 року SQL став стандартом для всіх професійних реляційних СКБД та став вбудовуватись у всі існуючі системи.

Основні переваги мови SQL:

- міжнародний стандарт мови;
- незалежність SQL, всі програми можна перенести з однієї СКБД в іншу з мінімальними доробками;
- допускається використання як в локальному режимі, так і для великих багатокористувацьких систем;
- таблична структура РБД добре зрозуміла, тому SQL э простим для вивчення ;
- в інтерактивному режими можна отримати рез-т запита за дуже короткий час без написання програми;
- SQL можна легко використати в додатках ;
- За допомогою SQL БД можна представити таким чином що різні групи користувачів будуть бачити різні відображення даних;
- Можливість динамічної зміни та розширення БД без зміни тексту програми;
- Підтримка архітектури клієнт-сервер.

4.2 Структура мови SQL.

SQL містить декілька розділів:

1) **DDL (Data Definition Language) -** мова визначення структур і обмежень цілісності баз даних. Сюди відносяться команди створення і видалення баз даних, створення, зміни та видалення

таблиць, представлень(віртуальних таблиць), індексів, керування користувачами та інше.

2) **DML (Data Manipulation Language) - оператори** маніпулювання даними:

DELETE – знищує рядки в таблиці (завжди виконується коректно, якщо зберігаються умови цілісності БД);

INSERT – вставити рядок, або перенести один чи кілька рядків з однієї таблиці в іншу;

UPDATE – обновити рядок чи поле у відповідності їз заданими умовами.

3) **оператор, для організації запитів до БД**

SELECT – вибрати рядки, оператор, який замінює всі оператори реляційної алгебри і дозволяє сформувати результуюче відношення, що відповідає запиту.

4) **Засоби керування транзакціями**

COMMIT – завершити комплексну взаємопов'язану обробку інформації, поєднану в транзакцією;

ROLLBACK – відмінити зміни, проведені в ході виконання транзакції;

SAVEPOINT - зберегти проміжний стан БД, помітити його, щоб можна було до нього повернутись.

Типи даних SQL.

Числові:

SMALLINT – цілі числа від -128 до +127
INTEGER – цілі числа від -32768 до +32767
BIGINT – цілі числа від -2 147 483 648 до +2 147 483 647
FLOAT – дійсні числа від $3.4*10^{-38}$ до $3.4*10^{38}$ з 7 значущими цифрами
DOUBLE PRECISION – дійсні числа від $1.7*10^{-308}$ до $1.7*10^{308}$ з 15 значущими цифрами
NUMERIC[(*розмірність* [, *точність*])] – фіксований формат, *розмірність* – загальне число знаків (максимальне 18 знаків); *точність* – число знаків після коми.
DECIMAL[(*розмір* [, *точність*])] – фіксований формат.

Дата, час:

DATE – поле дати, задається у форматі dd.mm.yyyy
TIME – час, задається у форматі hh:mm:ss
TIMESTAMP – дата і час, задається у форматі dd.mm.yyyy hh:mm

Текстові:

CHAR(*розмір***) [CHARACTER SET** *код***] [(COLLATE** *код1***)]** – рядок символів фіксованої довжини, що містить будь-які друковані символи розмірністю від 0 до 32767.
VARCHAR(*розмір*) **[CHARACTER SET** *код***] [(COLLATE** *код1***)]** – рядок символів змінної довжини, що містить будь-які друковані символи розмірністю від 0 до 32767.
BLOB [SUB_TYPE *число***] [SEGMENT SIZE [***розмір***]]** – поле, що містить дані великого об'єму такі як графіка, текст, цифровий звук, у двійковому вигляді. Якщо *число* дорівнює **1**, то це текст. Замість числа **1** можна вказувати константу **TEXT**.

Опис типу даних для стовпця типу **CHAR, VARCHAR** або **BLOB-текст** може включати пропозицію **CHARACTER SET** визначаючи специфічне кодування для вибраного стовпця. Інакше стовпець використовує визначену за замовчуванням для бази даних кодування. Якщо кодування бази даних змінене, всі стовпці згодом визначені мають нове кодування, але існуючі стовпці не змінюються.

За допомогою опції **COLLATE** вказується вибраний порядок сортування. Для кодування **WIN1251** допустимі порядки сортування **WIN1251, WIN1251_UA, PXW_CYRL. PXW_CYRL** встановлює порядок сортування для баз даних PARADOX.

4.3.Структура запитів SQL

Основним оператором для організації запиту SQL є SELECT. Один той самий запит може бути реалізований різними способами, вони можуть істотно відрізнятись по часу виконання, це особливо відчутно для великих БД. Запит –це команда, яка звертається до бази даних, та повідомляє її, щоб вона відобразила певну інформацію з таблиць в оперативну пам'ять. Всі запити формуються однією командою, повний набір опцій якої наступний:

SELECT[ALL | DISTINCT] <Список полів >
FROM< Список таблиць >
[**WHERE** < Предикат – умова вибірки чи з'єднання >]
[**GROUP BY** < Список полів >]
[**HAVING** < Предикат – умова для групи >]
[**ORDER BY** < Список полів, за якими впорядковане виведення>]

Пояснимо детально кожну частину команди:
ALL – в результуючий набір включається всі рядки, що відповідають умові відбору, тобто в результуючий набір можуть попасти рядки, що дублюються;
DISTINCT – тільки різні рядки, без повторень.
***** – всі стовпці вихідної таблиці,
FROM – перелік вихідних таблиць запиту;
WHERE – умови відбору рядків рез-ту, чи умови з'єднання кортежів(записів) вихідних таблиць (операція з'єднання в РА).
GROUP BY список полів групування
HAVING - задає предикат – умови, що накладаються на групу.
ORDER BY – список полів, впорядкування результуючого набору(наприклад: прізвище, № з/п).

У виразах **WHERE** можуть бути використані наступні предикати {=,<>,<,>,>=,<=}, **BETWEEN A and B**, **IN**(множини), **NOT IN**(множини), **IS NULL**(порівняння з невизначеним значенням) **IS NOT NULL, EXIT** та **NOT EXIT** – (існує та не існує). **LIKE, NOT LIKE.**

Оператор **LIKE** використовується для порівняння символьних даних відповідно до шаблону. Шаблон задається за допомогою одинарних лапок. Дозволяється використання наступних групових символів:

- знак підкреслення _ заміняє один символ;
- знак відсотків % заміняє будь-яку послідовність символів;

Розглянемо приклади використання оператора **SELECT**.
Нехай в нас є база даних **UNIVER**, що складається з таблиць

Таблиця **VUKLAD –** містить інформацію про викладачів та має наступну структуру

Назва поля	Тип поля	Примітки
VNOM	INTEGER	Код викладача, **первинний ключ**
VFAM	VARCHAR(15)	Прізвище
VIMA	VARCHAR(15)	Ім’я
VOTCH	VARCHAR(15)	По батькові
KAF	VARCHAR(4)	Назва кафедри
POSADA	VARCHAR(20)	Посада
OKLAD	FLOAT	оклад

Таблиця **PREDMET –** містить інформацію про предмети та має наступну структуру

Назва поля	Тип поля	Примітки
PNOM	INTEGER	Код предмета, первинний ключ
PNAME	VARCHAR(20)	Назва предмета
VNOM	INTEGER	Код викладача, зовнішній ключ
GOD	INTEGER	Кількість годин, зо відведена на вивчення предмета
SEMESTR	INTEGER	Семестр, в якому предмет вивчається

Таблиця **STUDENTS**- містить інформацію про студента, та має наступні поля

Назва поля	**Тип поля**	**Примітки**
SNOM	INTEGER	Код студента,,первинний ключ
SFAM	VARCHAR(15)	Прізвище студента
SIMA	VARCHAR(15)	Ім’я
SOTCH	VARCHAR(15)	По батькові
STIP	FLOAT	Розмір стипендії
GRUP	VARCHAR(5)	група
FORM	VARCHAR(8)	Форма навчання
FOTO	BLOB	фото

Таблиця **USPISH –** містить інформацію про результати складання іспитів

Назва поля	**Тип поля**	**Примітки**
NOM	INTEGER	Код, первинний ключ
DATA	DATE	Дата складання іспиту
PNOM	INTEGER	Код студента, зовнішній ключ
PNOM	INTEGER	Код предмета. Зовнішній ключ
OCINKA	INTEGER	оцінка

Приклади:

1) Вивести прізвище, ім’я, по батькові студентів, що навчаються в 1-й групі

SELECT sfam, sima, sotch FROM students WHERE grup= ‘ПМ-1’

2) Вивести всю інформацію про викладачів з таблиці VUKLAD

SELECT * FROM vuklad

3) Вивести всю інформацію з таблиці USPISH про студентів що мають відмінні та добрі оцінки

SELECT * FROM uspish WHERE ocinka IN (4,5);

4) Додати новий запис в таблицю STUDENTS (при додаванні нових записів в таблицю, значення поля первинного ключа має бути задано обов’язково(snom)

INSERT INTO STUDENTS (snom, sfam) VALUES (123,'Радько')

5) Вставити дані в таблицю sesiya з таблиці student, в цьому випадку в таблиці sesiya має бути хоча б один запис, імена полів не повинні співпадати

INSERT INTO sesiya (snomer, fam)
SELECT snom, sfam FROM students

7) Знищити запис з таблиці студент

DELETE FROM students WHERE snom =123

8) Змінити значення поля стипендія для студента з вказаним номером

UPDATE students SET stip=700 where snom =11

9) Наступний приклад демонструє, як можна відкоригувати структуру таблиці: додати нове поле в таблицю vuklad, тип поля визначений доменом NAM, що мав бути створений попередньо

ALTER TABLE vuklad ADD pole1 nam;

10) Додати нове поле в таблицю vuklad, поле типу CHAR

ALTER TABLE vuklad ADD polecko TYPE char(25);

4.4. Складні запити. Об'єднання в таблицях

Використання агрегатних та групових функцій в SQL

В SQL агрегатні функції повертають одинарне значення, обчислене за набором значень в стовпці. До таких функцій відносяться:

1) **COUNT(ім'я_поля)** – підрахунок кількості рядків або не-NULL значень поля.
   Допускає використання DICTINCT, ALL або *.
   COUNT(DICTINCT *ім'я_поля***)** – підрахунок не-NULL значень поля, виключаючи дублікати.
   COUNT([ALL] *ім'я_поля***)** – підрахунок не-NULL значень поля, включаючи дублікати.
   COUNT(*) – підрахунок кількості рядків, включаючи NULL значення полів.
2) **SUM(***ім.'я_поля***)** – розраховує суму усіх значень вибраного поля;
3) **AVG(***ім.'я_поля***)** – знаходить середнє арифметичне значень вибраного поля;
4) **MAX(***ім.'я_поля***)** – знаходить максимальне значення серед значень вибраного поля;
5) **MIN(***ім.'я_поля***)** – знаходить мінімальне значення серед значень вибраного поля.
6) **FIRST(***ім.'я_поля***)** – знаходить перше значення з серед значень вибраного поля.
7) **LAST(***ім.'я_поля***)** – знаходить останнє значення з серед значень вибраного поля.

Скалярні функції

Скалярні функції повертають значення, що базується на введеному значенні

1) **UPPER (***ім'я_поля* **)** – перетворює символьний рядок до верхнього регістра;
2) **CAST (***ім'я_поля* **AS** *тип_даних***)** – перетворює дані поля до певного типу даних.
3) **EXTRACT(***частина* **FROM** *ім'я_поля***)** – вибирає інформацію про дату і час з даних типу **DATE**, **TIME**, **TIMESTAMP**.

Частина – може бути **YEAR** (значення – 0-5400), **MONTH** (1-12), **DAY** (1-31), **HOUR** (1-23), **MINUTE** (1-59), **SECOND** (0-59.9999), **WEEKDAY** (0-6, 0 – неділя , 1 – понеділок і т.д.), **YEARDAY** (1-366).

Тип даних, які є результатом функції – **SMALLINT**, для секунд – **DECIMAL**(6,4).

4) **LEN() -** повертає кількість символів текстового поля
5) **ROUND() -** закруглює числове значення до вказаного в умові
6) **NOW() -** повертає поточну дату та час
7) **FORMAT() -** вказує в якому форматі мають виводитись даних

4.5. Об'єднання таблиць однакової структури

Оператор **UNION** використовується для об'єднання записів таблиць, які мають однакові структури, виключаючи дублікати. Для виведення усіх даних разом з **UNION** використовується оператор **ALL.**

Приклад:

1) Вивести прізвища, імена і по батькові усіх студентів і викладачів.

```
SELECT SFAM, SIMA, SOTCH FROM STUDENTS
UNION ALL
SELECT VFAM, VIMA, VOTCH FROM VYKLAD;
```

Встановлення зв'язків між таблицями

При організації запитів, що відбирають дані з кількох таблиць потрібно виконати їх об'єднання по ключових полях. Такі об'єднання можна виконати кількома способами. Перший полягає у

порівнянні значень ключових полів в умові **WHERE:**

Приклад:

2) Вивести прізвища студентів і їх оцінки.

```
SELECT STUDENTS.SFAM, USPISH.OCINKA
FROM STUDENTS, USPISH
WHERE STUDENTS.SNOM=USPISH.SNOM;
```

Другий спосіб полягає у встановленні об'єднань між таблицями по ключових полях:

INNER JOIN – встановлює об'єднання, в якому вибираються тільки ті записи, які містять співпадаючі значення в полях зв'язку;

Приклад: Вивести прізвища студентів та оцінки, що вони отримали.

```
3) SELECT STUDENTS.SFAM, USPISH.OCINKA
   FROM STUDENTS INNER JOIN USPISH
   ON STUDENTS.SNOM=USPISH.SNOM;
```

LEFT [OUTER] JOIN – встановлює лівостороннє зовнішнє об'єднання, тобто таке, в якому вибираються всі записи з лівої таблиці і записи з правої таблиці, для яких значення поля зв'язку співпадає із значенням поля зв'язку лівої таблиці;

Приклад: Вивести прізвища викладачів та назви предметів, які ті читають

```
4) SELECT VYKLAD.VFAM, VYKLAD.VIMA, VYKLAD.VOTCH,
   PREDMET.PNAME FROM VYKLAD LEFT JOIN PREDMET
   ON VYKLAD. PNOM = PREDMET. PNOM
```

RIGHT [OUTER] JOIN – встановлює правостороннє зовнішнє об'єднання, тобто таке, в якому вибираються всі записи з правої таблиці і записи з лівої таблиці, для яких значення поля зв'язку співпадає із значенням поля зв'язку правої таблиці;

FULL [OUTER] JOIN – створює повне зовнішнє об'єднання, в якому вибираються всі значення із лівої і правої таблиць(декартів добуток).

III. Вкладені запити.

Вкладені запити створюються шляхом розміщення одного

запиту (підзапиту) в середині другого. Підзапит повертає завжди одне значення.

Приклад:

8) Вивести номера дисциплін і оцінки, вищі від середньої.

SELECT PNOM, OCINKA
FROM USPISH
WHERE OCINKA > (**SELECT AVG**(OCINKA) **FROM** USPISH);

У вкладених запитах можуть використовуватись оператори **EXIST**, **ANY**, **ALL**, **SOME**, які в якості аргументу використовують підзапит.

Результатом оператора **EXIST** є значення „істинно", якщо він здійснює будь-яке виведення записів, і „хибно" в противному випадку. Підзапит, який використовується в якості аргументу оператора **EXIST,** може повертати декілька значень або жодного. Якщо він повертає хоч одне значення , то **EXIST** дає в якості результату – ІСТИНА і виконується зовнішній запит.

Приклад:

9) Вивести дані з таблиці USPISH, якщо в ній є відмінні оцінки.

SELECT * **FROM** USPISH **WHERE** USPISH.OCINKA=5 **AND EXIST**
(**SELECT** * **FROM** USPISH **WHERE** USPISH.OCINKA=5);

10) Вивести інформацію про студентів, які здали кілька іспитів. Цей запит можна виконати кількома способами

a) **SELECT** DISTINCT FIRST. SNON, FIRST. SFAM, FIRST. SIMYA
FROM STUDENTS FIRST, USPISH SECOND
WHERE EXIST
(**SELECT * FROM** USPISH THIRD WHERE
SECOND.SNOM=THIRD.SNOM AND SECOND.PNOM<>THIRD. PNOM)
AND FIRST.SNOM= SECOND.SNOM

b) **SELECT** * **FROM** STUDENTS FIRST WHERE EXIST

(**SELECT** * **FROM** USPISH SECOND WHERE FIRST.SNOM= SECOND.SNOM AND 1 < (**SELECT** COUNT (*) **FROM** USPISH WHERE USPISH.SNOM= SECOND.SNOM))

Існує ще три оператор, що орієнтовані на підзапити – це ANY, ALL та SOME. Але вони відрізняються від EXIST тим, що використовуються разом з реляційними операторами в підзапитах. Оператори ANY та SOME взаємозамінні, тому в поданих прикладах будуть працювати однаково. Оператор ALL вибирає всі значення, що виведенні підзапитом.

Оператори ANY та SOME взаємозамінні і в наших прикладах будуть працювати однаково. Оператор ANY може використовувати інші реляційні оператори за виключенням «дорівнює.», і таким чином реалізовувати порівняння, які не доступні, наприклад, в IN.

Приклади:

11) Вивести інформацію про студентів, що отримали оцінки з різних навчальних предметів.

SELECT * **FROM** STUDENTS **WHERE** SNOM = **ANY** (**SELECT** SNOM **FROM** USPISH);

12) Вивести дані про тих студентів, чиї оцінки вище або рівні отриманим 13.06.2015

SELECT * **FROM** USPISH **WHERE** OCINKA>= **ALL** (**SELECT** OCINKA **FROM** USPISH **WHERE** DATA='13.06.2015')

13) Вивести назви дисциплін, для яких існує хоча б одна дисципліна з кількістю годин менше ніж в поточної

SELECT PNAME **FROM** PREDMET **WHERE** GOD > (**SELECT** GOD **FROM** PREDMET)

Взагалі, запит, який був реалізований з допомогою ANY, ALL, може бути сформований з допомогою EXIST, але не навпаки.

4.6. Відображення та їх організація

Типи таблиць, з якими ми до сих пір мали справу, називаються *базовими таблицями.* Це таблиці, що містять дані. Але є інший вид таблиць – *відображення*. *Відображення(VIEW)* – це таблиця, вміст якої вибирається чи отримується з інших таблиць. Вони працюють в запиті чи в операторах DML, так само як і основні таблиці, але не містять ніяких власних даних. Відображення подібні до вікон через які здійснюється перегляд інформації, яка фактично зберігається в базовій таблиці. Відображення запит, який виконується щоразу, коли запускається відповідна команда. Результати виконання стають вмістом відображення. Створюються відображення командою

CREATE VIEW *ім'я_відображення* **[(***ім'я_поля* **[,** *ім'я_поля* **...])]** **AS <select> [WITH CHECK OPTION];**	**CREATE VIEW Vidm** **AS SELECT *** **FROM student** **WHERE stip** =1200:
CREATE VIEW KISP(SNOM, PNOM, OCINKA) **AS** **select** snom, pnom, ocinka **from** uspish	**CREATE VIEW** KLient_misto **AS** **SELECT * FROM** Klienty **WHERE** city = 'Рівне'

Це відображення можна тепер використовувати, як будь-яку іншу таблицю. До неї можна сформувати запит, модифікувати, знищувати, з'єднувати з іншими таблицями та відображеннями. Відображення завжди відображає самі останні дані вибрані з таблиці. Це чудовий спосіб надати публічний доступ до деякої інформації в таблиці. Насправді відображення є запитом, текст команди якого зберігається в БД, але для користувача нічім не відрізняється від базової таблиці.

Відображення може змінюватись командами DML, але модифікації не будуть впливати на саме відображення, всі вони будуть переадресовані до базової таблиці. За допомогою відображення на екран будуть виведені самі остання інформація, що була введена в базу даних.

UPDATE Vidm

SET stip = 1600
WHERE snom= 11

Зміни, що були внесені оператором UPDATE будуть збережені в базовій таблиці. Але зміни можна робити лише для тих записів і полів, що були відібрані запитом, визначеним для відображення. Тобто в нашому прикладі лише для студентів, стипендія яких більше ніж 600.

До створених відображень можна формувати запити:

SELECT * FROM VIDM WHERE SFAM LIKE 'K%';

Це еквівалентно виконанню наступної команди

SELECT * FROM student WHERE stip = 1100 **and** sfam **like** 'K%';

Для відображення можна задавати назви полів, відмінні від тих що використовуються в базових таблицях. Нова назва задається в дужках (в нашому прикладі розрахункове поле). Якщо не задати нові назви, команда працювати не буде..

CREATE VIEW KISP3(N, NN)
AS
select pnom, count(pnom) from uspish group by pnom ;

Сформуємо запит до нового відображення

select * from kisp3 where nn>2;

Відображення може бути засноване відразу не кількох базових таблицях. Наприклад: вивести прізвища, імена, назви предметів та оцінки, отримані кожним студентом

CREATE VIEW V2(FAM, IM, PR, OC)
AS
select student.sfan, student.sima, predmet.pnam, uspish.ocinka
from student, predmet, uspish where
student.snom=uspish.snom and predmet.pnom=uspish.pnom:

На основі такого відображення можна тепер будувати різноманітні запити. Наприклад: сформувати відомість результатів іспиту з певного предмету

select FAM, IM, OC from V2 where PR=' ЧММФ';

При створенні представлені можна використовувати також підзапити. Вивести оцінки, що отримані з деякого предмету і вищі ніж середній бал з того ж предмету:

CREATE VIEW AVGOC
AS
select *
from uspish first where ocinka >
(select avg(ocinka) from uspish second where second.pnom= first.pnom);

Знищити відображення можна командою
DROP VIEW <*ім'я відображення*>
При цьому базові таблиці залишаться без змін.

Не всі відображення дозволяють модифікацію даних. Деякі з них використовуються лише для читання при запиті. Загалом команди INSERT, UPDATE, DELETE можуть застосовуватись для відображень, але існує кілька правил, що визначають чи є відображення таким, що дозволяє модифікувати дані в базових таблицях.

Критерії, що визначають чи модифікує відображення дані наступні:

- відображення має засновуватись лише на одній базовій таблиці:
- воно має містити первинний ключ цієї таблиці:
- воно не повинно містити полів, які є агрегатними функціями:
- воно не повинно містити DASTINCT у визначенні:
- відображення не повинно містити **group by:**
- бажано, щоб не використовувались підзапити:
- воно може використовуватись в інших відображеннях, але тільки в таких що дозволяють модифікацію:
- воно не повинно містити константи, рядки або вирази серед полів виведення.

Різниця між відображеннями, що виконують модифікацію даних та відображеннями лише для читання не випадкові. Цілі для яких вони

створюються різні - відображеннями, що дозволяють модифікацію, використовуються в основному як базові таблиці. Користувач, як правило, не може навіть усвідомити, чи то є базова таблиця чи відображення. Вони, в основному, використовуються для захисту базових таблиць. Відображення тільки для читання дозволяють створювати та зберігати цілий арсенал складних запитів, використовуючи їх знову і знову. З рештою, їх можна використовувати в інших запитах.

Наведемо приклади;

1. Кількість всіх оцінок, що отриманні всіма студентами в той чи інший день

```
 CREATE VIEW P1 (udat,col)
     AS
select. DATA, count(*)  from uspish group by  DATA ;
```

2. Оцінки з певного предмета

```
CREATE VIEW P2
     AS
select. *, from uspish where pnom=203;
```

Ці відображення дозволяють модифікацію

3.Збільшення стипендії вдвічі – відображення тільки для читання, через присутність **STIP*2**

```
 CREATE VIEW P3 ( FAM, IM,  NSTIP)
     AS
     select sfan, sima, STIP*2
from student WHERE STIP=300;
```

4.

```
CREATE VIEW VIDM
AS
select * from student
where stip>600 ;
UPDATE Vidm SET stip = 700 WHERE snom=12;
```

Коли ми вносимо зміни в таблицю через запит, на екрані відобразяться лише ті запити, що задовольняють умові відбору відображення, і тому зміни базової таблиці можуть не відобразитись на екрані, іноді це може стати проблемою для користувача, бо він цього не бачить. Для виключення подібних моментів існує фраза WITH CHECK OPTION у визначенні

відображення. Ця фраза дозволить регулювати модифікації таблиці, тобто зміни будуть відбуватись лише в записах відібраних умовою відбору, всі інші просто ігноруватимуться

CREATE VIEW VIDM
AS
select snom, sfam from student
where stip=600
WITH CHECK OPTION;
Знищувати та корегувати різки такого відображення цілком можливо, а от введення нового запису неможливе. Але якщо створити інше відображення

CREATE VIEW NOWVIDM
AS
select * from VIDM;

то виконати команду вставки цілком можливо
INSERT INTO N NOWVIDM VALUES (212, 'Решетило ', 600)

Одна з сильних сторін SQL - здатність працювати з усіма рядками таблиці, що задовольняють мові відбору, але це не завжди зручно, коли SQL взаємодіє з іншими мовами програмування і потрібно передати результати виконання запиту в змінні, адже нам наперед не відомо кількість записів що будуть оброблятись. Вирішення цієї проблеми здійснюється за рахунок курсора . *Курсор* – це вид змінної, яка пов'язана із запитом. Він повинен бути оголошений раніше, ніж буде використаний. Це робиться командою
DECLARE CURSOR CSTOP
FOR
select sfan, sima, stip
from student WHERE STIP=600;

Запит, що оголошений в курсорі не буде виконуватись миттєво, адже представлено лише ого визначення. Коли в програмі потрібно виконати цей запит, це робиться наступним чином

OPEN CURSOR CSTOP

Значення курсору впорядковані, є перший, другий, та останній рядки, передача значень в курсор відбувається саме після виконання команди **OPEN.**

Команда FETCH використовується, щоб дістати значення першого вибраного рядка в змінні пам'яті, наприклад

FETCH CSTOP INTO: STUDFAM, :STUDIM, :STUDSTIP

Як правило, команду FETCH розташовують в циклі, для того щоб обробити всі вибрані запитом рядки. Якщо у FETCH немає більше записів, він буде виводити останнє значення до кінця циклу
Команда CLOSE CURSOR звільняє курсор від вибраних записів та закриває його. В нашому випадку буде виглядати наступним чином

CLOSE CURSOR CSTOP

Контрольні запитання та завдання до розділу 4

1. Чи можна SQL назвати повноцінною мовою програмування?
2. Назвіть основні переваги мови SQL.
3. Які агрегатні функції використовуються в SQL?
4. Яким оператором організовуються запити до бази даних в SQL?
5. Які види об'єднань в запитах вам відомі?
6. Скільки значень повертає вкладений запит?
7. Яким чином встановлюється зв'язок між двома таблицями в запитах?
8. Які оператори використовуються в вкладених запитах?
9. Що таке представлення?
10. Чим представлення відрізняються від запитів?

РОЗДІЛ 5. СТВОРЕННЯ Т ВИКОРИСТАННЯ STORED-ПРОЦЕДУР

5.1. Створення та використання процедур в SQL

Клієнтські додатки можуть звертатись до потужності SQL-сервера для розв'язування складних задач Збережені процедури, це підпрограми, що виконуються на сервері(в середині серверного процесу), створюються та зберігаються в БД. Таки процедури можуть виконуватись в клієнтських додатках в будь-який кількості, можуть маніпулювати даними в базі даних а також повертати клієнту результати свого виконання. Пишуться таки процедури на мові SQL. Кожна така процедура компілюється при першому виконанні, в процесі компіляції будується оптимальний план її виконання. Створюються процедури командою

CREATE PROCEDURE *ім'я_процедури*
[(*вхідний_параметр тип* [, *вхідний_параметр тип* ...])]
[**RETURNS** (*вихідний_параметр тип* [, *вихідний_параметр тип*...])]
AS
тіло_процедури;
[*термінатор*]

Синтаксис збереженої процедури складається з двох частин: заголовка і тіла. Заголовок включає команду **CREATE PROCEDURE** ім'я процедури, список вхідних параметрів і список параметрів, які повертаються з процедури. Деякі з цих складових можуть бути відсутні. Тіло процедури може включати DML-SQL-конструкції, а також, програмні конструкції FOR SELECT…DO, IF-THEN, WHILE...DO та інші.

Конструкції мови SQL

1. **Коментар**

   /*коментар***/**

2. **Конкатенація**

   'симв_рядок1'||'симв_рядок2'

3. **Опис локальних змінних**

DECLARE VARIABLE <*ім'я змінної*> <*тип змінної*>

4. **Оператор умови IF ... THEN ... ELSE ...**

   IF (*умова***)**
   THEN *оператор_1*
   [ELSE *оператор_2***];**

умова – логічний вираз, який має значення TRUE або FALSE;
оператор_1 – виконується, якщо умова приймає істинне значення. Замість *оператор_1* може бути блок операторів, обмежений конструкцією **BEGIN … END**; *оператор_2* – виконується, якщо умова хибна. Замість *оператор_2* може бути блок операторів, обмежений конструкцією **BEGIN … END**.

5. Оператор циклу WHILE … DO …

WHILE (*умова***) DO** *оператор;*

умова – логічний вираз, значення якого перевіряється перед кожним виконанням циклу;
оператор – оператор чи блок операторів **BEGIN … END**, який виконується, якщо умова приймає істинне значення.

6. Оператор SELECT

SELECT_команда INTO *список_змінних;*

Синтаксис даного оператора подібний до синтаксису команди **SELECT.** На відміну від звичайної команди **SELECT** результатом даної команди є один рядок, отримані дані записуються у змінні, вказані після ключового слова **INTO.**

7. Оператор циклу FOR … DO …

FOR SELECT_команда INTO *список_змінних* **DO** *оператор;*

SELECT_команда отримує дані з бази даних. Дана команда отримує за один раз тільки один рядок, і результат записується у відповідні змінні, вказані після ключового слова **INTO.** Перед кожною змінною після **INTO** ставиться двокрапка (**:**) для того, щоб відрізнити ім'я стовпця від імені змінної;
оператор – оператор чи блок операторів **BEGIN … END**, який виконується для кожного рядка, отриманого командою **SELECT.**

В тілі процедури може використовуватись оператор EXIT. Він здійснює вихід з циклу і перехід на останній END в процедурі. Команда SUSPEND икористовується тільки в процедурах, що повертають результати в точку виклику.

Існує два види збережених процедур: **SELECT** і **EXECUTE**. **SELECT**-процедури обов'язково повертають одне або декілька значень(рядків) і можуть використовуватись при створенні запитів разом з таблицями і представленнями.

EXECUTE-процедури можуть повертати 1 або не повертати жодного значення у точку виклику.
SELECT-процедури і EXECUTE-процедури описуються однаково, але викликаються по різному.
SELECT-процедури викликаються за допомогою конструкції:

SELECT * FROM *ім'я_процедури* [(*фактичний_параметр* [, *фактичний_параметр*…])];

EXECUTE - процедури викликаються за допомогою конструкції:

EXECUTE PROCEDURE *ім'я_процедури* [(*фактичний_параметр* [, *фактичний_параметр*…])]

Приклади:

1. Збільшити значення поля kurs на 1(при переведення на новий курс)

```
SET TERM ^ ;
CREATE PROCEDURE PR_1
as
begin
 update student set kurs=kuer+1 ;
 end^

SET TERM ; ^
```

Зверніть увагу, що в середині процедури використовується «;» - розділювач команд. Як відомо «;» є стандартним розділовим знаком для команд і сигналом для інтерпретатора SQL, що команда ведена повністю і можна приступати до її виконання. Тому, якщо створювати процедури з допомогою SQL- скриптів, необхідно перед створення процедури змінювати розділювач команд
SET TERM ^ ;. В кінці процедури записується протилежна команда SET TERM ; ^

2. Проіндексувати поле стипендія на задане число для певних студентів

```
SET TERM ^ ;

CREATE PROCEDURE PR2 ( ssum decimal(15,2), kf integer)
as
begin
 update student set stip=stip*:kf
 where stip=:ssum ;
end^

SET TERM ; ^
```

В наведеному прикладі використовується змінна :ssum, яка є вхідним параметром процедури, тут символ «:» використовується для тог щоб відрізнити змінні від назв полів, Двокрапка пере іменем змінної використовується лише в
SQL-операторах, у всіх інших операторах програми ці змінні використовуються без «:».

1. Проіндексувати поле стипендія для студентів, що здали іспит з певного предмету, знищити інформацію про деякого студента

```
CREATE PROCEDURE NEW_PROCEDURE
(    kf integer,    pnom_pm integer,     snom_pm integer)
as
begin
 if (exists(select snom,pnom from uspish
 where snom=:snom_pm and pnom=:pnom_pm))
 then
  begin update student set stip=:kf;
delete from uspish
where snom=:snom_pm and pnom=:pnom_pm ;
end
```

2. Вивести на екран інформацію про предмет, на вивчення якого відводиться задана кількість годин

```
      CREATE PROCEDURE PR3
      returns (  pn varchar(20),  g integer)
      as
      begin
  for select pnam,god from predmet where god=162
  into
 :pn,:g
  do
       suspend;
      end^

SET TERM ^ ;
```

3. Процедура нарахування стипендії

```
CREATE PROCEDURE NEW_STIP
returns (  sn integer,  st decimal(15,2), avg_bal decimal(5,1))
as
begin
for select snom, avg(ocinka) from uspish group by snom
```

```
into :sn,:avg_bal
do
begin
if avg_bal >=4  then
 update student set stip=720
 where student.snom =:sn ;
else
if avg_bal <4  then
update student set stip=null
suspend;
end;
end^
```

Обробка виключень, повідомлення про помилки

Однією з особливостей мови збережених на сервері процедур і тригерів Interbase є можливість використовувати так звані виключення. Виключення Interbase багато в чому схожі на виключення інших мов програмування високого рівня, проте мають свої особливості. Фактично виключення Interbase - це повідомлення про помилку, яке має власне ім'я і текст повідомлення, що задається програмістом. Створюється виключення наступним чином.

CREATE EXCEPTION <*ім'я виключення*> <*текст виключення*>;

Виключення легко видалити або змінити: видалення здійснюється командою

DROP EXCEPTION < имя_исключения >

ALTER EXCEPTION <имя_исключения> < текст виключення >.

Щоб використовувати виключення в процедурі, що зберігається, або тригері, необхідно скористатися командою наступного вигляду.

EXCEPTION < ім'я виключення >;

Давайте розглянемо використання виключень на простому прикладі процедури, що зберігається. Нехай процедура виконує ділення одного числа на інше і повертає результат. Нам необхідно відстежити випадок ділення на нуль і активувати виключення, якщо дільник дорівнює нулю.

Створимо виключення

CREATE EXCEPTION zero_divide ' Cannot divide by zero!'

яке буде генеруватись процедурою

CREATE PROCEDURE SP_DIVIDE (DELIMOE DOUBLE PRECISION, DELITEL DOUBLE PRECISION)

```
RETURNS (
RESULT DOUBLE PRECISION)
AS BEGIN
if (Delitel<0.0000001) then BEGIN
EXCEPTION zero_divide; Result=0; END ELSE BEGIN
Result=Delimoe/Delitel; END
SUSPEND; END
```

У виключеннях було б мало користі, якби не було можливості обробляти їх на рівні бази даних. Для цього використовується конструкція

```
WHEN EXCEPTION
<ім'я_виключення> DO BEGIN
/*обробка
виключення*/
END
```

Використання цієї конструкції допомагає уникнути стандартного повідомлення про помилки(того, що генерується СКБД) і виконати власні дії по обробці помилки.

Наступний приклад демонструє використання виключень. Припустимо, в нас є процедура sp_test_except, що викликає нашу процедуру SP_DIVIDE. Звичайно, приклад досить надуманий, але він демонструє зміст використання виключень в SQL.

```
CREATE PROCEDURE sp_test_except (Delitel DOUBLE
PRECISION) RETURNS (rslt DOUBLE PRECISION, status
VARCHAR (50)) AS
BEGIN
  Status='Everything is Ok';
  SELECT result FROM sp_divide(12,:Delitel) INTO :rslt;
  SUSPEND;
  WHEN EXCEPTION
    Zero_divide DO BEGIN
Status='zero value found!';
rslt=-l; SUSPEND; END
END
```

1. ***Генератори та тригери***

Генератор – це механізм, який створює послідовний унікальний номер, що автоматично вставляється в стовпець під час таких операцій, як **INSERT** або **UPDATE**. Генератори зазвичай використовуються для створення унікальних значень, які можуть бути вставлені в стовпець, який використовується як первинний ключ.

Створення генератора

CREATE GENERATOR *ім'я_генератора*;
При створенні генератора за замовчуваннчм його початкове значення дорівнює нулю.

Ініціалізація генератора

SET GENERATOR *ім'я_генератора* **TO** *ціле_число***;**

Функція GEN_ID

GEN_ID (*ім'я_генератора* **,** *ціле_число***);**
Дана функція збільшує поточне значення генератора на ціле число.

Приклад:

Створити генератор, який використовується для створення унікальних значень поля SNOM, який є первинним ключем таблиці, починаючи з 101.
CREATE GENERATOR GEN_STUDENTS;
SET GENERATOR GEN_STUDENTS **TO** 100**;**
INSERT INTO STUDENTS **(**SNOM**,** SFAM**)**
VALUES(GEN_ID(GEN_STUDENTS**,1),** 'IBAHOB'**);**

Тригери в InterBase - це особливий вид процедури, що зберігається, яка виконується автоматично при вставці, видаленні або модифікації запису таблиці чи представлення (view). Тригери можуть "спрацьовувати" безпосередньо до або відразу ж після вказаної події.

Як відомо, SQL дає можливість нам вставляти, видаляти і модифікувати дані в таблицях бази даних за допомогою відповідних команд - INSERT, DELETE і UPDATE. Погодьтеся, що було б непогано мати можливість перехопити передавану команду і що-небудь зробити з даними, які додаються, віддаляються або змінюються. Наприклад, записати ці дані в спеціальну табличку, а заразом записати, хто і коли зробив операцію над цією таблицею. Чи відразу ж перевірити дані, що вставляються, на яку-небудь хитру умову, яку неможливо реалізувати за допомогою опції CHECK, і залежно від результатів перевірки прийняти зміни, що проводяться, або відхилити їх; змінити дані на основі якого-небудь запиту в одній або кількох пов'язаних таблицях. Ось для того і існують тригери.

На відміну від процедур, що зберігаються, тригер ніколи нічого не повертає (та і нікому повертати, адже тригер явно не викликається). З тієї ж причини він не має вхідних параметрів, але замість них має контекстні змінні NEW і OLD. Ці змінні дозволяють отримати доступ до полів таблиці, до якої приєднаний триггер.

Тригер завжди прив'язаний до якоїсь певної таблиці або представлення і може "перехоплювати" дані тільки з цього об’єкта. Він виконує, так би мовити, роль віртуального «цензора», що дозволяє або відхиляє зміни, аналізує похибки, або може в разі необхідності «доповісти куди треба».

1. Створення тригера

CREATE TRIGGER *ім’я_тригера* **FOR** *ім’я_таблиці*
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION *номер***]**
AS *тіло_тригера*
термінатор

[ACTIVE | INACTIVE] – необов’язковий параметр, який визначає дію тригера після завершення транзакції. **ACTIVE –** тригер використовується (по замовчуванню). **INACTIVE –** тригер не використовується.
{BEFORE | AFTER} – обов’язковий параметр, який визначає коли відбудеться активізація тригера до події (**BEFORE**) чи після (**AFTER**).
{DELETE | INSERT | UPDATE} – визначає операцію над таблицею, яка викликає тригер для виконання.
[POSITION *номер***] –** визначає порядок виклику тригера для обробки однієї події, наприклад при додаванні запису в таблицю. Номер має бути цілим від 0 до 32 767, по замовчуванню дорівнює нулю. Тригер, який має менший номер, виконується раніше.
тіло_тригера – складається з двох частин:
1) блок опису локальних змінних, які використовуються в тригері. Кожна змінна описується конструкцією
DECLARE VARIABLE *ім’я_змінної тип_змінної* **;**
і закінчується крапкою з комою (;);
2) блок програмного коду, який починається оператором **BEGIN** і завершується оператором **END**
BEGIN
команди_InterBase
END
Кожна команда завершується крапкою з комою (;).

У блоці програмного коду тригера використовуються дві контекстні змінні: **OLD** і **NEW**. Змінна **OLD.***ім'я_стовпця* відповідає за старі значення стовпця, змінна **NEW**.*ім'я_стовпця* – за нові значення.
До виконання команди, дані з таблиці заносяться в буфер пам'яті, де тригер має до них доступ через змінні **OLD** і **NEW та** може дозволити чи заборонити дії над ними.
Розглянемо існуючи види тригерів -**Active (inactive) before(after)**

Дані, що прислані на вставку поміщаються в буфер та можуть бути переглянуті або змінені з допомогою контекстної змінної NEW, до полів цього запису можна звертатись, як до полів запису.

```
CREATE TRIGGER Table_ex FOP Table_example
ACTIVE BEFORE  INSERT POSITION 0
AS BEGIN
IF (NEW.ID IS NULL) THEN
NEW.ID=GEN_ID (GEN_TABLE_EXAMPLE_ID,1)
END
```

Ми бачимо, що в даному тригері використовується змінна NEW. Загалом існують деякі умови використання цих двох змінних. Так, контекстна змінні OLD не може бути використана в тригерах before(after) insert, а змінна NEW в тригерах before(after) delete. Але обидві ці змінні можуть використовуватись в тригерах before(after) update

Приклад. Створити генератор **GEN_STUDENTS** і створити тригер **STUDENTS_BI**, який використовує даний генератор як лічильник для поля **SNOM** при додаванні записів у таблицю **STUDENTS**.

```
CREATE GENERATOR GEN_STUDENTS;

SET TERM ^ ;

CREATE TRIGGER STUDENTS_BI FOR STUDENTS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.SNOM IS NULL) THEN
    NEW.SNOM = GEN_ID(GEN_STUDENTS,1);
END ^

SET TERM ; ^
```

Модифікація тригера

```
ALTER TRIGGER ім'я_тригера
[ACTIVE | INACTIVE]
[ {BEFORE | AFTER} {DELETE | INSERT | UPDATE} ]
```

[POSITION *номер***]**
[AS *тіло_тригера* **]**
[*термінатор***]**

При модифікації можна змінити статус тригера, порядок і спосіб його виконання, внести зміни у тіло тригера. За замовчуванням тригер створюється активним, якщо ж зробити його неактивним він не буде використовуватись при виконання операції. Це буває корисним при виконанні деяких позапланових операцій над даними або при виправлені даних вручну. Не варто через тригер змінювати дані в таблиці до якої він прив'язаний не через контекстні змінні, а через оператори SQL, бо це може привести до за циклювання і аварійній зупинці сервера InterBase.

Приклад:

Змінити статус активного тригера *TR1* на пасивний.

ALTER TRIGGER TR1 **INACTIVE;**

2. Видалення тригера

DROP TRIGGER *ім'я_тригера***;**

Якщо база даних досить складна, є досить велика ймовірність виникнення помилок на будь-якому етапі коригування даних, коли, наприклад, при вставці даних в головну таблицю запускаються ЗП, що активізують тригери і вставку даних у пов'язані таблиці, тощо. При появі помилок InterBase повідомить про це і виконає відкат змін у таблицях Але клієнтський додаток може відслідкувати помилку і підтвердити транзакцію, якщо це допустимо. Можна виконати обробку помилки безпосередньо в тілі тригера з допомогою конструкції WREN…DO. Так само як і ЗП тригер може генерувати обробку виключень, тому що він є різновидом ЗП.

РОЗДІЛ 6. ТРАНЗАКЦІЇ

6. 1. Транзакції та цілісність бази даних

Щоб зрозуміти що собою представляє транзакція, розглянемо приклад. Нехай існує база даних, в який зберігається інформація про деяку торгову фірму, що постачає продукти в супермаркети. Потрібно змінити номер продукту з 13 на 20. Для цього потрібно провести зміни в 4-х взаємозв'язаних таблицях.

```
UPDATE  Продукти        UPDATE      Склад
SET     ПР = 20         SET     ПР = 20
WHERE   ПР = 13;        WHERE       ПР = 13;

UPDATE  Поставки        UPDATE      Наявність
SET     ПР = 20     SET     ПР = 20
WHERE   ПР = 13;        WHERE       ПР = 13;
```

Цей приклад показує, що єдина(суцільна), з точки зору користувача, операція може потребувати кількох операцій над БД. Крім того між виконанням цих операцій може навіть порушуватись цілісність даних, наприклад, в ній можуть бути записи про поставки, для яких немає записів про відповідні продукти. Протиріччя зникне лише після виконання всіх операцій на зміну номера.

Отже, *транзакція - це послідовність операцій, які переводять базу даних з одного узгодженого стану в інший, але не гарантує збереження цілісності БД в проміжних операціях*.

Ніхто крім користувача, що генерує транзакцію не може знати, про те коли може виникнути неузгоджений стан в БД, і після якої операції він зникне, тому в мові SQL є спеціальні команди, з допомогою яких можна визначити, що транзакція завершена, або виконати так званий відкат транзакції.

Властивості транзакцій. Способи завершення транзакцій.

На даний час розрізняють: плоскі чи класичні транзакції, ланцюгові транзакції, вкладені транзакції.

Плоскі транзакції характеризуються 4-ма класичними властивостями:

- *Атомарністю* – виражається в тому, що транзакція повинна бути виконана повністю або не виконана зовсім;
- *Узгодженість* – гарантує, що транзакція не порушує взаємної узгодженості даних;
- *Ізольованість* – означає, що конкуруючі за доступ до БД транзакції обробляються послідовно, ізольовано один від одної, але для користувачів це виглядає ніби вони виконуються паралельно.

- *Довговічність* – якщо транзакція завершена успішно, то ті зміни, які були виконані, не можуть бути втрачені ні за яких обставин (навіть у випадку збоїв в системі).

Можливі два варіанти завершення транзакцій: якщо всі операції виконані успішно і не відбулося ні яких збоїв транзакція фіксується (завершення оператором COMMIT); до тих пір поки транзакція не зафіксована БД можна перевести (повернути) до попереднього стану (завершення оператором ROLLBACK відкат транзакції).

Для успішного завершення транзакції потрібне завершення всіх її операторів. Якщо сталось щось таке, що робить неможливим завершення транзакції, потрібно перевести БД до попереднього стану. В стандарті SQL для цього використовуються оператори COMMIT та ROLLBACK. Стандарт визначає, що транзакція починається з першого SQL-оператора, що задається користувачем, або програмно, всі наступні оператори складають тіло транзакції. Транзакція завершується одним з 4-х можливих шляхів:

1) Оператор COMMIT означає успішне завершення, і відображення зміни в БД;
2) ROLLBACK перериває транзакцію, відміняє зміни в рамках цієї транзакції; нова транзакція починається безпосередньо після використання ROLLBACK;
3) Успішне завершення програми, в якій була ініційована поточна транзакція, означає успішне завершення ніби був виконаний оператор COMMIT;
4) Помилкове завершення програми перериває транзакції ніби був виконаний оператор ROLLBACK.

В цій моделі кожний оператор, що змінює стан БД, розглядається як транзакція.

В подальших версіях в СКБД була реалізована розширена модель транзакцій в ній використовуються наступні 4 оператора:

- BEGIN TRANSACTION – повідомляє про початок транзакції;
- COMMIT TRANSACTION – повідомляє про успішне завершення та фіксує всі зміни;
- SAVE TRANSACTION – створює всередині транзакції точку збереження, яка відповідає проміжному стану БД, збереженому на момент виконання цього оператора. В цьому операторі може стояти ім’я точки збереження, в ході виконання транзакцій може бути кілька точок збереження, що відповідає кільком проміжним станам.

– ROLLBACK має дві модифікації. Якщо він використовується без додаткового параметра, то він інтерпретується як оператор відкату всієї транзакції. Якщо він має параметр і зустрічається у вигляді ROLLBACK B, то він інтерпретується як оператор часткового відкату в точку збереження B.

Доцільно використовувати точки збереження в довгих і складних транзакціях.

Реалізація в СКБД принципу збереження проміжних станів забезпечується журналом транзакцій. Він призначений для забезпечення надійного збереження даних в БД, можливість відновлення узгодженого стану бази після будь-яких збоїв. При цьому потрібно дотримуватись наступних правил:

- результати зафіксованих транзакцій повинні бути збережені у відновленому стані БД;
- результати незафіксованих транзакцій повинні бути відсутніми у відновленому стані БД.

.

Відновлення БД потрібно проводити в таких випадках:

- індивідуальний відкат транзакцій;
- відновлення після втрати даних з ОП (м'який збій);
- відновлення після поломки зовнішнього носія (жорсткий збій).

Для відновлення узгодженого стану БД при індивідуальні й відкатці транзакцій потрібно знищити наслідки операторів модифікації БД, які виконували ці транзакції. При відновленні при м'якому збої необхідно відновити вміст журналу транзакцій, що зберігаються на дисках. При жорсткому збої – відновити вміст БД по архівних копіях та журналах транзакцій з неушкоджених дисків.

У всіх трьох випадках основою відновлення є надлишкове збереження даних. Ці надлишкові дані зберігаються в журналі, що містить послідовність записів про зміни в БД. Можливі два варіанта ведення журналу транзакцій:

1.Для кожної транзакції підтримується окремий локальний журнал зміни в БД (локальний журнал, для індивідуальних відкатів і можуть підтримуватись в ОП). Крім того підтримується загальний журнал змін БД, який використовується. для відновлення стану БД після м'яких та жорстких збоїв. Цей підхід дозволяє швидко виконувати індивідуальні відкати, але приводить до дублювання інформації в загальному та локальному журналах. Тому частіше використовують другий варіант

2. Ведення лише загального журналу змін БД, який використовують також при виконанні індивідуальних відкатів.

Розглянемо саме цей варіант: загальна структура журналу може бути представлена у вигляді деякого послідовного файлу в якому фіксуються всі зміни в БД, які відбуваються в ході виконання транзакцій. Всі транзакції мають свої внутрішні номери, тому в журналі фіксуються всі зміни, що виконуються всіма транзакціями. Кожний запис в журналі транзакцій помічається № транзакції, до якої він відноситься та значенням атрибутів, які він міняє. Крім того для кожної транзакції фіксується команда початку та кінця транзакції. Для більшої надійності журнал транзакцій часто дублюється системними засобами СКБД.

Журналізація змін тісно пов'язана з буферизацією сторінок в ОП. Якби запис при будь яких змінах в БД реально записувалась би в зовнішню пам'ять, це привело б до істотного сповільнення роботи. Тому запис в журналі також буферизується і чергове оновлення переноситься в зовнішню пам'ять при повному заповненні буфера. Якщо відбувається м'який збій то вміст буферів втрачений.

Для проведення відновлення БД необхідно мати деякий узгоджений стан журналу і БД у зовнішній пам'яті. Тобто запис про зміни об'єкта БД повинна попадати в зовнішню пам'ять раніше ніж змінений об'єкт виявиться у зовнішній пам'яті БД. Відповідний протокол називається Wribe Abead Log (WAL) – „пиши спочатку в журнал ".

Розглянемо тепер, як може виконати операції відновлення СКБД, якщо в системі підтримується журнал у відповідності з протоколом WAL.

Для того, щоб індивідуальний відкат транзакцій був можливий, всі записи в журналі по даній транзакції зв'язуються в зворотній список. Відкат можливий лише для незакінчених транзакцій. Він виконується наступним чином:

- вибирається остання операція;
- виконується протилежна їй за змістом операція;
- будь які з протилежних операцій теж заносяться в журнал, щоб у випадку м'якого збою можна було виконати відновлення;
- при успішному завершенні відката в журнал заноситься запис про це.

На момент м'якого збою можливі наступні стани транзакцій:

- транзакція успішно завершена (COMMIT) і для всіх операцій отримано підтвердження в зовнішній пам'яті;
- транзакція успішно завершена (COMMIT), але для деяких операцій не отримано підтвердження в зовнішній пам'яті;
- транзакція отримала і виконала команду відката ROLLBACK;
- транзакція незавершена.

Є два підходи: використання тіньового підходу або журналізація посторінкових змін в БД.

1) При відкритті файлу таблиця відображає номер логічних блоків у фізичні адреси блоків зчитується в ОП. При модифікації блоку у зовнішній пам'яті виділяється новий блок. При цьому поточна таблиця в ОП міняється, а тіньова в зовнішній пам'яті не змінюється. При збої в ОП в зовнішній пам'яті зберігається стан БД до відкриття.
2) Періодично виконуються операції встановлення точки фізичної узгодженості БД і тіньова таблиця в зовнішній пам'яті змінюється. При відновленні просто зчитується тіньова таблиця. Недолік – велике завантаження зовнішньої пам'яті.

Відновлення після жорсткого збою: основою для відновлення є журнал та архівна копія.

Відновлення починається з копіювання архіву по журналу в прямому напрямку виконуються всі операції; для незакінчених транзакцій виконується відкат. Якщо журнал втрачений – лише архівна копія. Архівують БД при переповненні журналу. В журналі вводиться так звана „жовта зона", коли всі транзакції тимчасово припиняються, незавершені закінчуються, база приводиться в узгоджений стан. Можна проводити архівацію.

2. Якщо з БД працює одночасно кілька користувачів, то кожна транзакція повинна виконуватися так, ніби вона ізольована. Таке виконання називається паралельним. Є кілька шляхів розпаралелювання запитів.

Горизонтальний паралелізм – виникає тоді, коли БД розподілена на кількох фізичних пристроях збереження (кількох дисках). При цьому інформація з одного відношення розбивається на частини по горизонталі. Цей вид паралелізму називають розпаралелюванням або сегментацією даних. Паралелізм досягається шляхом виконання однакових операцій (напр. фільтрації) над різними даними. Ці операції можуть виконуватись паралельно різним процесам, вони не залежні. Результат виконання цілого запиту складається з результатів виконання окремих операцій. Час виконання такого запиту при відповідному сегментуванні даних істотно менше, чим час такого ж запиту традиційним способом одним процесом.

Вертикальний паралелізм – досягається конвеєрним виконанням операцій, що відповідають запиту користувача. Цей підхід вимагає серйозного вдосконалення в моделі виконання реляційних операцій ядром СКБД. Він передбачає, що ядро СКБД може провести декомпозицію запита, базуючись на його функціональних компонентах при цьому ряд підзапитів може виконуватись паралельно, з мінімальним зв'язком між окремими кроками виконання запита.

Дійсно, якщо ми розглянемо послідовність операцій RA

R5=R1[A,C] – проекція

R6=R2[A,B,D] – проекція
R7=R5[A>128] – фільтрація
R8=R5[A]R6 – умовне з'єднання

то 1 і 3 операції можуть об'єднати і виконати паралельно з 2, а потім виконати 4.

Загальний час такого виконання буде менший ніж при звичайному виконанні.

Третій вид паралелізму є гібридом 2-х перших. Ці види паралелізму використовують в додатках, це може істотно скоротити час виконання складних запитів над великими об'ємами пам'яті.

При паралельній обробці транзакцій виникають деякі проблеми. Вони умовно ділться на 4 типи:

1. **втрачені зміни.** Ця ситуація може виникнути, коли дві транзакції змінюють один і той же запис. Приклад: перший оператор прийняґв замовлення на продаж 30 моніторів (на складі є 40); 2-й – 20 моніторів; 2-й update 20 шт; 1-й update – 10 шт. БД в неузгодженому стані, та ще 10 моніторів числяться но складі.
2. **проблема проміжних даних.** Той самий приклад. Припустим 1-й оператор оформляє замовлення, в БД внесені зміни залишок 10 моніторів, Але замовник ще обговорює деталі з оператором. В цей час 2-й оператор намагається прийняти замовлення на 20 моніторів, але бачить, що на складі залишилось всього 10, оператор віджмовляє клієнту. В цей час 1-й клієнт передумав купляти монітори, 1-й оператор виконує відкат транзакцій, на складі знову 40 моніторів. Замовник втрачений, але було б гірше коли оператор продав 10 моніторів заніс 0 в БД; після цього1-й оператор заніс 40 на склад, хоча 10 з них продано. Така ситуація стала можливою, тому що 2-й оператор мав доступ до проміжних даних.
3. **проблеми неузгоджених даних.** Припустимо обидва оператори починають роботу в один час і отримують початковий стан БД 40. 1-й оператор змінив транзакції, 2-й отримує нове значення – 10. 2-й оператор буде вважати, що порушена цілісність його транзакції.
4. **проблема рядків привидів.** Припустим оператор готує два звіти детальний та розширений. Детальний вже підготовлений, а коли готується розширений, оператор проводить транзакцію, другий зовсім на співпадає з першим, хоча бази знаходяться в узгодженому стані.

3. Для того, щоб уникнути подібних проблем, потрібно виконати деяку процедуру узгодженого виконання паралельних транзакцій. Ця процедура повинна задовольняти наступним правилам:

- в ході виконання транзакцій користувач бачить лише узгоджені дані, користувач не повинен бачити проміжних неузгоджених даних.
- Коли дві транзакції виконуються паралельно, то СКБД гарантовано підтримує принцип незалежного виконання тразакцій, який проголошує, що результати виконання такими ж, якби спочатку виконувалась транзакція 1 а потім транзакція 2.

Така процедура називається серіалізацією транзакцій. Фактично, вона гарантує, що кожен користувач (програми), звертаючись до БД, працює з нею так, ніби не існує інших користувачів, що одночасно з нею звертаються до даних.

Для підтримки паралельної роботи транзакцій будується спеціальний план. План (спосіб) виконання транзакцій називається серіальним, якщо результат одночасного виконання транзакцій еквівалентний результату деякого послідовного виконання цих транзакцій.

Найбільше розповсюдження механізмом виконання серіальних транзакцій є механізм блокування. Самий простий варіант блокування об`єкта на весь час виконання транзакції. Якщо дві транзакції працюють з 3-ма таблицями T1, T2, T3. в момент звертання транзакцій, таблиця блокується об`єктом, що до неї звернувся, 2 інші транзакції знаходяться в стані очікування.

Недолік: затримка виконання транзакцій.

Між двома паралельними транзакціями існують наступні типи конфліктів:

- W-W – транзакція 2 намагається змінити об`єкт, змінений незакінченою транзакцією 1;
- R-W – транзакція 2 намагається змінити об`єкт, прочитаний назакінченою тоанзакцією 1;
- W-R – транзакція 2 намагається читати об`єкт, змінений незакінченою транзакцією 1.

Практичні методи серіалізації транзакцій засновані на врахуванні цих конфліктів.

Блокування (сінхронічніі захоплення) можуть бути застосовані до різного типу об`єктів. Найбільшим об`єктом може бути вся БД, але цей вид блокуавння робить БД недоступною для всіх додатків. Інший вид блокування таблиць, цей вид блокування кращий, тому що дозволяє паралельно працювати з іншими таблицями. В деяких СКБД блокування виконується на рівні сторінок. Цей вид ще більш гнучкий, тому, що дозволяє працювати з однією таблицею кілька транзакцій, з різними сторінками.

Для підвищення паралельності виконання транзакцій використовують комбінування різних типів синхронізованих захватів.

- сумісний режим блокування – нежорстоке, розділене блокування, позначають S (Shared). Цей режим означає розділений захват обєкта і вимагається для виконання операції читання об`єкта. Об`єкти заблоковані даним типом блокування, не змінюються у ході виконання транзакцій і дотупні іншим лише в режимі читання.
- Монопольний режим блокування – жорстке, або ексклюзивне блокування, X (exelusive). Передбачає монопольне захоплення об`єкта і вимагає для здійснення операції занесення, знищення, модифікації. Об`єкти, що заблоковані даним типом блокування, фактично в монопольному режимі обробки та недоступний для інших.
- H – жорстке блокування

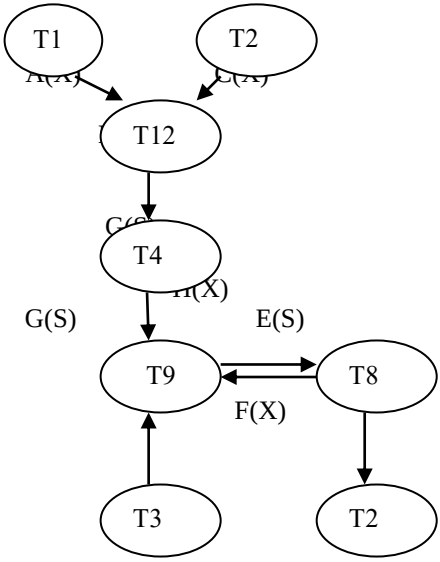
Захоплення об`єктів кількома транзакціями по читанню сумісні. (кілька транзакцій може читати один об`єкт). Захоплення одного об`єкта 1 транзакцією по читанню, другою по запису несумісні.

На жаль, застосування різних типів блокувань приводить до проблеми **тупиків**. Приклад: нехай транзакція А спочатку жорстко блокує Т1, а потім жорстко блокує Т2; а транзакція В навпаки. Якщо дві транзакції почали роботу одночасно, то А буде безкінеченно очікувати розблокування Т2, а В навпаки. Ситуації можуть бути більш складними.

Основою виявлення тупікової ситуації є побудова графа очікування транзакцій. Це направлений граф в вершинах якого розташовані імена транзакцій. Якщо транзакція А очікує закінчення транзакції В, то з вершини в вершину іде стрілочка.

Нехай граф побудований для транзгакцій Т1, Т2, ... , Т12, які працюють з об`єктами БД А, В, …, Н.

S – нежортстке



.З діаграми видно, що T9, T8, T2, T3 утворюють цикл. Саме наявність циклу є признаком виненнення тупіка. Розпочинається починається з вибору транзакції-жертви. Критерієм вибору є вартість транзакцій. Жертвою вибирається сама дешева транзакція. Вартість транзакції визначається на основі багатофакторної оцінки: час виконання, число накопичених захоплень, пріорітет.

Після вибору транзакції-жертви виконується відкат цієї транзакції., звільняються захаплення і може бути продовжене виконання транзакції.

Рівні ізольованості користувачів пов`язані з проблемами, які виникають при паралельному виконанні транзакцій. Всього введено 4 рівні ізольованості користувачів.

Самий високий відповідає протоколу серіалізацій, це рівень SERIALIZABLE. Цей рівень забезпечує повну ізоляцію транзакцій і повну коректну обробку паралельних транзакцій.

Наступний рівень – рівень підтивердженого читача – REPEATABLE READ. На цьому рівні транзакція на має доступ до проміжних чи кінцевих результатів інших транзакцій. Тому такі проблеми, як пропавші очікування, проміжні чи неузгоджені дані виникнути не можуть. Але можуть бути побачити наступний записв, добавлений іншим користувачем, залишиться проблема рядків-призраків. Цю проблему краще розвязувати алгорітмічно, виключаючи повторне виконяння запиту в одній транзакції.

Наступний рівень ізольованості пов`язаний з підтвердженням читанням він називається READCOMITED. На цьому рівні транзакція не має доступу до проміжних результатів інших транзакцій;транзакції не можуть обновити рядок вже поновлену іншиою транзакцією.

Самий низький рівень ізольованості називається рівень непідтвердженого (брудного) читання READUNCOMMITED.

Транзакція бачить проміжні і неузгоджені дані, та рядки-призраки.

В стандарті SQL2 існують операції задання рівня ізольованості виконання транзакцій.

SET TRANSACTION IZEOLATION LEVEL [{SERIALIZABLE
REPEATABLE READ/
READ COMMITED/
READ UNCOMITED}] [{READ WRITE/
READ ONLY}]

Гранульовапні сінхронізаційні захоплення. При виконанні захопленькрупних об`єктів, ймовірність конфліктів зростає. В більшості сучасних СКБД – покортежне блокування. Але було б нерозумно застосовувати покортежне блокування при знищені всіх рядків відношення.

Це привело до поняття гранульованого сінхронного захоплення і розробці відповідного механізма.

Синхронічні захоплення можуть бути застосовані по відношенню до різних об`єктів; файлів, відношеннь і кортежів. Рівень об`єкта визначається тим, яка операція виконується (при знищені відношення - захоплення відношення, при знищені кортежа – кортеж). Об`єкти будь-якого рівня може бути захоплений в режимі S (розподільному) або X (монопольному). Вводиться спеціальний протоколгранульованих захоплень і визначаються нові типи захоплень. Об`єкт більш високого рівня повинен бути захоплений в режимі IS або IX або SIX.

IS (INTENED FOR SHARED LOCK) – запобігаєрозподільному блокуванню. По відношенню до деякого складного об`єкта. О означає бажання захопити даний об`єкт, що входить в нього в сумісному режимі.

IX (interned fox exclusive lock) – запобігає жорстокому блокуванню відношення до деякого складного об`єкту (при занченні IX)

SIX (shared, Internded for exclusive lock) розподіляє блокування, запобігає жорстокому блокуванню складових. Наприклад при переключанні відношення з можливим знищенням будь-яких кортежей, економічне захоплення відношення відносно в SIX, а файл в IS)

РОЗДІЛ 7. ТЕХНОЛОГІЇ ФІЗИЧНОГО ЗБЕРІГАННЯ ТА ДОСТУПУ ДО ДАНИХ

1. Основні етапи доступу до даних.

Для збереження даних не існує ідеального способу. Тому СКБД повинна мати кілька структур збереження даних для різних задач та частин системи. Процес пошуку даних полягає в наступному: спочатку в ОП комп'ютера читається частина записів з допомогою диспетчера файлів. Диспетчер файлів визначає сторінку на який розташований потрібний запис,, для читання цієї сторінки використовується диспетчер дисків, він визначає фізичне розташування сторінки і посилає до ОС запит на введення-виведення даних.

Основні функції диспетчера дисків – приховати від диспетчера файлів всі деталі фізичних дискових операцій введення-виведення даних. І замінити їх логічними операціями зі сторінками.

На початковому етапі БД не містить даних, але в ній є набір пустих сторінок. Для розташування записів з даними X ДФ створить набір сторінок та розташує на них дані X, подібним чином будуть створені

набори сторінок для даних Y, та Z. Отже буде створено 4 набори сторінок X,Y,Z, та набір порожніх сторінок. При додаванні записів до набору X буде ДФ витягне нову пусту сторінку та додасть її до набору, при знищення, звільнить і перенесе у набір вільних сторінок. Ті ж самі дії будуть виконані для наборів Y,Z. Отже немає гарантії, що логічно зв'язані сторінки будуть фізично розташовані поруч. Тому послідовність сторінок кожного набору вказують з допомогою вказівників. Заголовок кожної сторінки містить дані про фізичну адресу наступної логічної сторінки. Для визначення інформації про розташування різних наборів сторінок диспетчер дисків організує окрему сторінку з номером 0, яку часто називають таблицею розташування або сторінкою 0, де перелічені всі набори сторінок разом із вказівниками на першу сторінку кожного набору.

Якщо при знищенні записів на сторінці утворились пусті місця то ДФ перемістить всі записи на початок сторінки. Таким чином кожний запис ідентифікується за номером запису та місцем розташування його по відношенню до кінця сторінки. При зміщенні запису корегується лише місце розташування на сторінці номер запису залишається сталим, отже доступ по номеру запису є досить швидким. Якщо запис не поміщається на сторінці, то частина його розташовується на так званій сторінці переповнення та ідентифікаційний номер запису буде змінений. Для деякого файлу завжди можна здійснити послідовний доступ до записів, які там зберігаються.

Та крім послідовного доступу існують інші способи : хешування, індексування, використання ланцюжка вказівників технології стискання.

2. Поняття індексу, застосування індексів

Дані, що зберігаються в БД невпорядковані, і СКБД все одно де і як вони зберігаються. Два записи, що вводились одночасно можуть бути розташовані на різних сторінках БД. За таких умова, якщо пошук по базі даних виконувати простим перебором, то це може займати багато часу і ресурсів ПК. Тому для прискорення пошуку застосовують такий механізм як індексування.

Індекс - це впорядкований вказівник на записи деякої таблиці. Вказівник означає, що індекс містить значення одного або декількох полів таблиці, а також адресу сторінки де ці записи зберігаються. Тобто індекс складається з двох части: „значення поля " + „фізичне місце розташування ". Основна функція індексів забезпечити швидкий пошук записів в таблиці.

Індекси використовуються:

- Для прискорення виконання запиту. Індекси створюються для полів, які використовуються в умовах відбору SQL-запитів.

- Забезпечення унікальності значень поля первинного ключа. При додаванні нового запису виконується пошук по індексному файлу, що створений для поля первинного ключа, якщо не знайдено такого значення, то додається нове значення поля первинного ключа.

Часто виникає питання: чому б не створити індекси для всіх полів таблиці раз вини прискорюють виконання запитів. Але надмірне створення індексів запирає дисковий простір. Крім того при модифікації таблиць(вставка/знищення записів) відбувається перебудова індексів, що приводить до збільшення часу на виконання запиту до таблиці з індексами ніж до таблиці без індексів.

Існує ще так зване привило 20%:якщо запит на вибірку повертає більш ніж 20% даних з таблиці, то використання індексів може сповільнити вибірку.

Ще один момент використання індексів пов'язаний з оптимізатором запитів. Оптимізатор – це сукупність механізмів, які будують план виконання запиту. Коли користувач формує запит до БД, він вказує, що повинно бути виконано але не вказує як це зробити. На основі переданого запиту оптимізатор будує план виконання. Коли аналізуються умови на вибірку типу WHERE, ORDER BY оптимізатор намагається використовувати індекси. Нажаль це не завжди ті індекси, що є найбільш ефективними для даного запиту. Що також сповільнює час виконання запиту.

Третім випадком, коли не потрібно використовувати індекси є поля з обмеженим набором даних. Наприклад, якщо поле приймає значення «ч» і «ж» немає ніякого змісти використовувати індекс..

Три основних випадки, коли використання індексу прискорює виконання запиту:

- Коли це поле використовується в умовах пошуку в запиті;
- Коли з’єднання таблиць(join) використовує це поле;
- Коли це поле використовує фразуORDER BY.

3. Процедури індексування та хешування

Забезпечення цілісності посилань – при додаванні записів у пов’язані таблиці виконується перевірка чи існує відповідний запис у батьківській чи дочірній таблиці.

Часто виникає питання: чому б не створити індекси для всіх полів таблиці? Відповідь очевидна – це приведе до надмірного завантаження пом’яті ком’ютера та сповільненню виконання запитів. Так, наприклад, при додаванні/відніманні записів в таблиці, серверу приходиться перебудовувати індекси. Це дві основні причини, що перешкоджають загальній індексації.

Та є ще 2 зауваження. Перше – якщо запит відбирає 20% всіх записів таблиці, то існуючі індекси затримують виконання запит у.

Спробуємо розібратись з індексами. Припустимо нам потрібно вибрати інформацію про студентів, зо здали іспит з певного предмету. Запит заснований на інформації, що зберігається в 2-х файлах, які можуть зберігатись на різних сторінках. Припустимо, що у файлі предметів використовується список предметів, впорядкований за алфавітом.

PN(індекс)
Алгебра
Мат.аналіз
Програмування
ЧММФ

SNOM	PN	NAME	OCINKA
121	Мат.аналіз	Поляков	5
122	Алгебра	Козак	4
123	ЧММФ	Вус	4
124	Програмування	Гриценко	3
125	ЧММФ	Хомяк	3

Можливі два способи пошуку інформації про студентів, що здали іспит з ЧММФ.:

- Знайти файл записів про успішність і вибрати з нього всі, в якому є рядок з ЧММФ;
- Знайти файл з назвами предметів, в ньому ЧММФ, та за вказівниками знайти всі записи з файлу про успішність. Якщо доля всіх студентів що здали ЧММФ по відношенню до всіх інших записів невелика , то другу стратегія буде кращою, тому що пошук у файлі предметів буде швидшою, а потім і пошук по вказівникам пройде швидко.

Файл предметів називають ***індексом*** по відношенню до успішності, а файл успішності індексованим по предметам. Індексний файл є файлом особливого типу,: кожен запис складається з 2-х частин даних та вказівника на номер запису з такими даними. Якщо індекс заснований на полі первинного ключа він називається первинним, якщо на іншому полі – вторинним(в нашому прикладі - вторинний)..Індекс заснований на ключовому полі називається ***унікальним***. Недоліком використання індексів є сповільнення процесу додавання нових записів, адже при додаванні нового запису потрібно оновлювати ще й файл індекса, додаючи до нього запис новий вказівник.

Кожен файл може мати кілька індексів. Наприклад : успішність проіндексована за предметами та оцінками. Для пошуку записів потрібно буде здійснити 2 пошуки. Часто індекси будують на основі кількох полів. В такому випадку пошук записів можна здійснити за одним переглядом.

Отже, основною метою використання індексів є прискорення процесу пошуку зва рахунок зменшення операцій введення-виведення. Якщо індекс створений на основі первинного ключа, то не маж необхідності в індексному файлі зберігати вказівники на всі вказані записи, можна лише вказати максимальне значення ключа на сторінці та номер сторінки. Індекс з описаною структурою називається ***нещільним.*** Перевага нещільного індексу в тому, що розмір його невеликий, тож і пошук ведеться швидше. Недолік – не модна здійснити перевірку деякого значення.

Іншим способом впорядкування даних є хешування. Хешування називається процедура швидкого прямого доступу до записів на основі заданого значення деякого поля, при чому не обов'язково, що це поле було ключовим. Особливості цієї технології наступні:

- Кожен запис в БД зберігається за адресом (хеш-адресом), що розрахована на основі спеціальної хеш-функції, що заснована на значенні деякого поля(хеш-поля) даного запису;
- Спочатку розраховується адреса, а тоді ДФ розташовує за заданою адресою вказаний запис;
- Для пошуку запишу теж спочатку обчислюється хеш-адреса, а тоді ДФ посилає запит на читання потрібного запису.

Найпростіший приклад хеш-функції:

Хеш-адреса = залишок від ділення значення хеш-поля на просте натуральне число.

Теоретично для визначення хеш-адреси можна використовувати значення самого ключового поля, але практично діапазон значень ключових полів, може бути значно ширшим за діапазон можливих адрес. Отже, потрібно знайти таку хеш –функцію, що звузити діапазон до оптимальної величини з врахуванням можливості резервування додаткового простору.

Недоліками хешування є

1. фізична послідовність записів в середині збереженого файлу майже завжди відрізняється від послідовності ключового поля, або будь-якої іншої логічної послідовності, і між послідовно розміщеними записами можуть бути проміжки невизначеного розміру.

2. Часто може виникнути ситуація що кільком записам відповідає однакове значення хеш-функції, тоді хеш-функцію потрібно виправляти. Можна скористатись методом прямого перебору. Припустимо на деякий пустій сторінці можна розмістити записів, що мають однакову хеш-адресу. Пошук в межах сторінки можна вести простим перебором. Але n+1 запис потрібно вже буде розташовувати на додатковій сторінці переповнення. При збільшенні розмірів файлу, збільшується і час пошуку та ймовірність спів падіння значень хеш-функції. Цю проблему можна вирішити шляхом перебудови хеш-функції та завантаження файлу з розширеною хеш-функцією. Щоб всі значення хеш-функції були унікальними можна використати в якості хеш-поля ключове поле..

Основні принципи методу розширеного хешування наступні:

- Якщо в якості хеш-функції використовується функція φ, а значення деякого ключового поля z дорівнює p, то значення псевдоключа буде φ(p). Псевдоключ - вказівник на місце збереження запису.:
- Файл,що зберігається буде містити пов'язаний з ним каталог. Що складається із заголовка який містить значення g –глибина каталогу, та 2^g на сторінки з кількома записами на них.

Тут був описаний лише один спосіб побудови хеш-функцій, їх є безліч.

Для виконання запитів можна використовувати і не менш ефективний спосіб ланцюжків вказівників. Обидва файли(файл вказівників та файл даних) при цьому можуть знаходитись в одному наборі. Файл вказівників називають батьківським, а записів –дочірнім.

Розглянемо вищезгаданий приклад

PN(індекс)
Алгебра
Мат.аналіз
Програмування
ЧММФ

SNOM	NAME	OCINKA
121	Поляков	5
122	Козак	4
123	Вус	4
124	Гриценко	3
125	Хомяк	3

В дочірньому файлі відсутнє поле з назвою предмета, але є вказівники на пов’язані записи, тобто на тих хто отримав оцінки по однакових предметах.

Переваги такої структури: простіше виконання операції вставки та знищення та менший розмір файлів на диску, ніж займає відповідна індексна структура. Але якщо провести запит на пошук предмету ,що здали студенти, то така структура не буде зручною, тут більше підійде індексна або хеш структура. Та якщо батьківська структура має велику кількість записів, то для неї теж прийдеться застосовувати індексну, або хеш структуру.. Вдосконалення батьківської структури можна було б провести за рахунок додавання нового вказівника в дочірній частині на пов'язаний запис в батьківський структурі. тоді можна уникнути перегляду всіх записів при пошуку.

3. Процедура стискання економить не лише місце на диску але й кількість операцій введення-виведення, тому що доступ до даних меншого розміру потребує меншої кількості операцій введення-виведення. Найбільш розповсюдженою технологією є технологія заснована на різницях, при який деяке значення заміняється на даними про його відмінності від попереднього. При такий технології дані потрібно зберігати послідовно і поруч, тому що при їх розпакуванні потрібно мати значення попередньої величини. Такий вид стискання ефективний для списків (вони зберігаються послідовно), в таких випадках вдається стиснути навіть вказівники. Один із способів стискання на основі різниці – видалення символів, що повторюються напочатку запису, із виказанням їх кількості, тобто переднє стискання

Студент	0 - Студент
Студентка	7- ка
Студентський	7-ський

Подібним чином виконується заднє стискання (видалення пробілів в кінці слова).

Інший спосіб = ієрархічне стискання: припустимо в деякому файлі. Завдяки кластеризації виконаній по полю PNOM з назвами предмету, окремо містяться всі записи . про студентів, що здали цей предмет, назва предмету записана один раз. Тобто всі дані будуть стиснуті в один ієрархічний запис.: назва предмета, та всі дані про студентів, що

здали цей предмет. Такий запис складається з 2-х частин: постійної з назвами предметів, і змінної(або групи повторень) – про студентів.

USPISH			
	SNOM	SNOM	OCINKA
Алгебра	3415	Котенко	5
Фізика	3412	Поляков	5
	3414	Грищенко	3
Хімія	3413	Старова	4

Подібне стискання можна використовувати для індексів, в яких кілька значень, що розташовані послідовно містять одинакові дані, але різні значення вказівника, або при між файловій кластеризації, коли записи про студента об'єднуються із записами про його оцінки по всіх предметах.

4. Створення індексів в SQL

Індексом називають впорядкований список полів чи груп полів в таблиці, це корисний інструмент, що використовується у всіх сучасних СКБД. Коли створюється індекс БД запам'ятовує відповідний порядок всіх значень цього поля. Перевагою використання індексів є прискорення пошуку відповідних записів. Але вони мають і недоліки - сповільнення виконання операцій модифікації таблиць, займають місце на диску. Тобто при створенні індексу потрібно приймати зважене рішення створювати його чи ні. Індекси можуть складатись з кількох полів, тоді головним є перше поле, друге впорядковується в середині першого, третє – другого і т.д.

CREATE [UNIQUE] [ASC[ENDING]| [DESC[ENDING]]

INDEX <ім'я індексу> **ON** <ім'я таблиці> **(**<ім'я поля>**[,**< ім'я поля **>]…)**

Зрозуміло, що таблиця вже має бути в пам'яті комп'ютера та мати відповідні поля. Наприклад, в таблиці студент найбільш ймовірним полем індексу може бути поле прізвище

CREATE INDEX SFAMIND ON STUDENT(SFAM)

Для створення унікальних індексів (без повторень значень поля) використовується ключове слово UNIQUE. Практично такий індекс є

первинним ключем таблиці, для таблиці студент таким індексом може бути SNOM

CREATE UNIQUE INDEX SNAMB ON STUDENT(SNOM)

Такий індекс не буде створений, якщо поле SNOM значення, що повторюються. Тому краще створювати індекси відразу після створення таблиці до введення в неї значень. При створенні первинного ключа на таблицю автоматично створюється унікальний індекс. Але не всякий унікальний індекс є первинним ключем. Первинні та унікальні індекси використовуються для забезпечення цілісності посилань. При створенні зовнішнього ключа теж автоматично створюється індекс, який використовується для перевірки чи інує відповідне значення у пов'язаній таблиці.

Індекси реалізовані у вигляді двійкового дерева і коли в таблицю додаються нові записи в дерево додаються нові гілочки. Причому додавання відбувається не в «хвіст», а на кінцях інших гілочок і з часом дерево стає розбалансованим і пошук по ньому сповільнюється. Тому індекси потребують періодичної перебудови. Перебудову індексу послідовно виконують дві команди

ALTER INDEX <name> INACTIVE;

ALTER INDEX <name> ACTIVE;

INACTIVE переводить індекс в неактивний стан і дерево можна перебудувати.

ALTER INDEX – має кілька обмежень, з його допомогою не можна перебудувати індекси. що використовуються в первинних зовнішніх та унікальних ключах, якщо вини в даний момент використовуються в запитах. Також для перебудови індексу потрібно мати права SYSDBA.

Перебудова індексу командами

DROP INDEX <ім'я індексу>

CREATE INDEX<ім'я індексу>

Призводить до створення індексу з «чистої сторінки». Але знову ж таки дійсні обмеження вказані вище для ALTER INDEX.

Третім способом перебудови індексу є резервне копіювання бази даних. При цьому, дані що входять ів індекс не зберігаються в резервній копії. Зберігаються лише визначення індексів. При відновленні з резервної копії індекс перестроюється заново.

Четвертий спосіб покращити ефективність індексу – зібрати статистику по індексам командою

SET STATISTICS INDEX <name>:

Статистика це величина в межах від 0 до 1, значення якої залежить від числа неоднакових записів в таблиці. Перерахунок статистик не перебудовує індексів, тому вільний від більшості обмежень, згаданих вище, і дає оптимізатору можливість оптимізатору прийняти вірне рішенні про використання індексу. Перерахувати статистику може системний адміністратор, або той, хто створив індекс.

Якщо індекс є комбінацією кількох полів, то значення кожного з них може повторюватись, але комбінація значень має бути унікальною.